7-17-2020 3:00 PM

Automated Anomaly Detection and Localization System for a Microservices Based Cloud System

Priyanka Prakash Naikade, *The University of Western Ontario*

Supervisor: Madhavji, Nazim H., *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Science degree in Computer Science

Abstract

Context: With an increasing number of applications running on a microservices-based cloud system (such as AWS, GCP, IBM Cloud), it is challenging for the cloud providers to offer uninterrupted services with guaranteed Quality of Service (QoS) factors. **Problem Statement:** Existing monitoring frameworks often do not detect critical defects among a large volume of issues generated, thus affecting recovery response times and usage of maintenance human resource. Also, manually tracing the root causes of the issues requires a significant amount of time. **Objective:** The objective of this work is to: (i) detect performance anomalies, in real-time, through monitoring KPIs (Key Performance Indicators) using distributed tracing events, and (ii) identify their root causes. **Proposed Solution:** This thesis proposes an automated prediction-based anomaly detection and localization system, capable of detecting performance anomalies of a microservice using machine learning techniques, and determine their root-causes using a localization process. **Novelty**: The originality of this work lies in the detection process that uses a novel ensemble of a time-series forecasting model and three different unsupervised learning techniques that avoid defining static error thresholds to detect an anomaly and, instead follow a dynamic approach. **Experimental Results:** The proposed detection system was experimented using different variants of ensembles, evaluated on a real-world production dataset out of which two proposed ensembles outperformed the existing static rule-based approach with average F1-scores of 86% and 84%, average precision scores of 82% and 77% and average recall scores of 91% and 93% respectively across 6 experiments. The proposed detection ensembles were also evaluated on the Numenta Anomaly Benchmark (NAB) datasets and results show that the proposed method performs better than the Numenta's standard HTM model score. **Research Methodology**: We adopted an agile methodology to conduct our research in an incremental and iterative fashion. **Conclusion**: The two proposed ensembles for anomaly detection perform better than the existing static rule-based approach.

Keywords

Cloud Computing, Microservices, Monitoring, Anomaly Detection, Distributed Tracing, Performance Anomalies, Unsupervised Machine learning, Time series, Localization.

Summary for Lay Audience

The stability of the cloud ecosystem is at stake with the continual growth and expansion of cloud adoption. For example, sluggish access to data, applications, and web pages frustrate the users and employees alike, and some performance problems can even cause application crashes and data loss. Existing monitoring frameworks often do not detect serious issues among a large volume of issues generated during the cloud system usage. This thus affects the recovery response times and effective use of maintenance human resource. We have developed an automated system to detect serious performance issues and their root causes that aimed to aid the maintenance team in fixing them.

The proposed detection system uses a novel combination of two algorithms to detect anomalies and it was evaluated on a real-world dataset from a production environment. The results show that two novel detection ensembles perform better than the existing static error thresholding approach. We also evaluated the proposed detection methods on the independent datasets with favourable outcomes.

Acknowledgments

First and foremost, I would like to express my sincerest gratitude to my supervisor, Dr. Nazim H. Madhavji, for giving me the opportunity to do master's program and research under him; for providing me with continuous guidance and motivation; and most importantly, for having faith in me throughout the ongoing challenges of research. He continually offered his support, time, and knowledge throughout these past two years, and always encouraged me to test my abilities, to keep striving for more. Above all, he is a very kind and understanding person. I'm thankful to the faculty and staff of the Department of Computer Science for providing all the support and facilities provided during my masters.

I am grateful to IBM (International Business Machines Corporation) for allowing me to do part of my research on their premises through an internship program. My sincere thanks to John Steinbacher and Jonathan Bennett for being an excellent supervising mentor at IBM who not only guided me throughout the internship but also in my master's research. Their appreciation for even the smallest work or updates motivated me to continue doing my work.

I would also like to extend my heartfelt appreciation to my wonderful research team for helping me throughout my research: Yasmen Wahba and Ahmed Ibrahim. As challenging as research can be at times, my team made it fun and boosted me with confidence whenever I felt low. I would also like to thank my fellow lab mates Ibtehal Noorwali, Elham Rahmani, Darlan Arruda, and others, who supported me when I was a fresher. Even with minimal interactions, watching my fellow mates strive indirectly taught me what perseverance and hard work can lead to; their achievements and patience inspired me a lot. Also, I would like to thank my family and friends for always being there to pat my back during my most stressful moments, for their support and encouragement throughout my master's program and life in general.

Finally, I would like to thank all the researchers out there for their incredible work and giving the world a hope that nothing is impossible if you dare to put forward your feet and proceed with transforming ideas into reality and not just let it stay in your mind.

Table of Contents

List of Tables

List of Figures

Glossary of Abbreviations

AD	Anomaly Detection
ML	Machine Learning
KNN	K-Nearest Neighbor
LOF	Local Outlier Factor
IF	Isolation Forest
LSCP	Locally Selective Combination of Parallel Outlier Ensembles
LSTM	Long Short-Term Memory
OC-SVM	One-Class Support Vector Machine
SLR	Systematic Literature Review
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
JSON	JavaScript Object Notation
ES	ElasticSearch

Glossary of Terms

Microservices	Microservices is a cloud-native architectural approach to develop software applications where a single application is composed of many loosely coupled and independently deployable smaller modules/components.
Distributed Tracing	Distributed Tracing is a method of understanding the flow of data as it propagates through the components of applications
True Positives	True Positives (TP) are the number of instances that are abnormal or anomalies (such as high response time of a system) and the model also predicts it as abnormal.
True Negatives	True Negatives (TN) are the number of instances that are normal points and the model also predicts it as normal.
False Positives	False Positives (FP) are the instances that are predicted as abnormal when they are normal data points.
False Negatives	False Negatives (FN) are the number of instances that are predicted as normal by the models but are abnormal data points.
Precision	Precision is the proportion of data points that were correctly predicted as anomalies over the total number of data points that were predicted as either anomaly or normal
Recall	Recall the proportion of data points that were correctly predicted as anomalies over the total number of anomaly points
F1-Score	F1-Score measure the harmonic mean of precision and recall
KNN	a density-based anomaly detection technique that assumes that the normal data exists around a dense neighborhood whereas abnormal data lies far away.
LOF	Local Outlier Factor (LOF) is an unsupervised density-based algorithm that relies on k-nearest neighbors and assumes that

	the density around an outlier will be significantly different than the density around its neighbors.
LSCP	LSCP is a framework used for the detection of outliers using an ensemble of unsupervised outlier ensembles
IF	Isolation Forest (IF) is an unsupervised learning technique that uses the concept of isolation and assumes that an anomaly can be easily separated in a few steps while the normal points which closer could take more steps to be separated.
OC-SVM	One Class SVM is an unsupervised learning technique based on the working of the Support Vector Machine (SVM), a supervised classifier that tries to find an optimal hyperplane having a maximum margin to separate two classes of data points.
LSTM	LSTMs are a special kind of RNNs that is capable of handling long-term dependencies in the sequential data.
Gradient Descent	It is an optimization algorithm that is used to find the values of parameters (weights, bias) of a function (f) that minimizes a cost function or loss.
Learning rate	The learning rate controls the size of steps or the amount of the weight adjusted with respect to the loss gradient.
Mahalanobis Distance	is a measure of the distance between a point P and a distribution D

Chapter 1

1 Introduction

This thesis focuses on the problem of anomaly detection and localization (i.e., determining the faulty component) for a microservices-based cloud environment. In this chapter, we describe the context of our topic followed by the issues associated with the Enterprise Cloud Applications and Cloud monitoring that motivated us to investigate this topic. We then overview the solution approach, its impact, novelty, contribution, and thesis outline.

1.1 Context

Cloud computing has come a long way since its inception in the 1950s [110]. It refers to the delivery of services such as servers, storage, databases, networking, software, etc., over the internet. In short, it means providing IT infrastructure to companies, firms, or individual end-users. Based on the different requirements of the companies or customers, the cloud offers three different types of services models: Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS) [104]. Enterprises and industries across various sectors have been using cloud services to meet their computing demands. Until recently, the prevalence of enterprise cloud computing was attributed to factors like low cost, safe environment, high capacity, etc. However, with the advancement of technologies such as the Internet of Things (IoT), Artificial Intelligence (AI), Blockchain,etc., there has been a tremendous increase in the growth of the cloud market [105]. A study conducted by International Data Group (IDG) shows that 73% percent of enterprises have at least one application or a portion of their infrastructure on the cloud. The global cloud market is expected to cross $600 billion by 2024 [7].

There has been an increasing demand from users and a rise in new architectural styles since the time cloud computing has taken the front stage across most of the IT infrastructure. Applications are required to be highly scalable and available on the cloud. In the past, most of the organizations adopted the monolithic architectural style, a traditional way of building applications, where individual components are combined into a single indivisible unit

which means all the functions are managed and served in one place. It comprises of client-side user interfaces, server-side application, and a database, where the server-side monolith application executes logic, handles HTTP requests, retrieves/updates information in the underlying databases. Hence, even smaller modifications would consume a considerable amount of time and rework in building and deploying the new version since any code change affects the whole system thus, making the development process longer [8]. Businesses require frequent updates to the application which costs time and resources for the organization [9]. The components cannot be scaled independently and since they are tightly coupled it is difficult to change the language, framework, or technology. The hindrances are not feasible for modern-day enterprises that are trying to keep pace with technological advancements happening now and then. Hence, in order to achieve agility, better scalability, speed, and reliability of applications, the modern-day enterprises are migrating to Microservices architecture [10,11,20].

According to a survey [12] by Nginx, Microservices are entering mainstream projects. About 70% of the organizations are either using or investigating microservices amongst which 29% of the businesses are using it in production, 16% are using it in development environments and 24% are evaluating them. Tech giants like Netflix, Amazon, e-Bay, Uber, and other small to medium enterprises have adopted microservices [17].

1.2 Research Motivation and Problem statement

Though microservices have several advantages, it becomes complex in terms of its management and monitoring due to its nature of the distribution. Maintainability is as crucial as the development of enterprise cloud applications. Firstly, with several enterprises hosting their applications on the cloud, the stability of the cloud ecosystem is at stake. The existing monitoring framework cannot handle and process such a huge volume of data collected from various applications or cloud systems. On the other hand, with the cloud-native microservices architecture, the more complex the software system gets, the harder it becomes to analyze and troubleshoot problems. Monitoring the health of the microservices-based cloud is essential in order to prevent the failure or degradation of the system. Occurrences of performance problems such as degraded performance (slow page response) and service downtimes (unreachable service endpoints) have become a norm

rather than an exception in a complex environment [15]. These degraded performances are an indication of possible future failures.

Performance issues have had a poor impact on revenues in the past. As per a survey conducted by the Aberdeen group, as additional one-second delay in page response can reduce page hits, user satisfaction, and sales by 11%, 16%, and 7% respectively. In 2010, Amazon had experienced 3-hour intermittent performance issues which resulted in a loss of $1.75m in revenue per hour [16]. It is to be understood that the magnitude of the impact of poor performance on revenue is proportionate to the operations of the organization. But it goes on to show that the impact can be quite significant in certain contexts.

Performance problems have a direct impact on the Quality of Service (QoS) factors, such as Reliability, Availability, Security, and others [16], as well. Prolonged performance degradation can affect the reliability and loyalty ratings of the services from the users. In March 2019, Facebook and Instagram suffered partial service outages which impacted consumers as well as the developers building the apps on the world's largest social network. In June 2019, multiple issues regarding Slack's degraded performance across all its services such as login, messaging, posting files, calls, and integrations with other apps were reported [18]. In 2019, Google Cloud Platform (GCP) experienced major issues with services such as cloud storage and dataflow, which affected multiple products, with major APIs getting affected globally. In July 2019, many iCloud users across the world received a "Service Unavailable – DNS failure" message for several hours. Though the issue was resolved, users still failed to use functions such as Find My iPhone [19].

Apart from the adverse impact on revenues and user satisfaction, developers and operations teams spend a significant number of hours in diagnosing the root-cause and fixing post disruptions of the services. With various performance issues being generated, several anomalies/defects go undetected affecting the system adversely by causing failures/outages. Secondly, the Operations team or the Site Reliability Engineers (SREs) are uncertain as to which defects/issues are to be resolved first and are required to spend a lot of time analyzing them. Thirdly, it is difficult to trace the root cause of the issues once identified/occurred requiring a huge amount of time and manual effort.

Hence, there is a dire need to proactively identify, prioritize and fix the issues in order to maintain the health of the cloud system and thereby achieve the IT operational excellence and avoid the Service Level Agreements (SLA) violations. Considering the scale and dynamism of microservices-based cloud system, there is a need to build an automated monitoring framework, generic to all services, which would be capable of detecting the performance issues/anomalies by exploiting the services' behaviour over time, identifying its root-cause and reporting them to the operations team.

1.3 Research Question

Information from the above section leads to the following important questions:

RQ1: How do we design an automated monitoring framework or anomaly detection system given the scale and dynamic nature of microservices?

RQ2: How to locate the root cause of the anomalies using the metric data from microservices?

RQ3: How to evaluate or validate the accuracy of the models given the lack of labeled data?

These questions are explored in the core chapters of the thesis as are their responses (Chapter 5, Chapter 6, and Chapter 7).

1.4 Solution Approach

Our research emphasizes on easing the monitoring process of a microservices-based cloud system by developing an automated prediction-based anomaly detection and localization system. The system performs two critical functions:

(i) It detects performance anomalies using a time-series deep learning model and an ensemble of unsupervised learning techniques. It can handle a huge volume of data generated from each of the individual microservices and it avoids the burden of defining static thresholds used in existing monitoring framework and other literature works as discussed in Chapters 3 and 4.

(ii) It identifies the casual components of the detected anomalies, which would enable the DevOps or IT operations team to fix them so as to maintain a healthy cloud environment.

For this purpose, trace events were collected to understand the communication or flow of transactions across microservices in the third-party cloud system. A high-level system was designed, based on initial analysis. Iteratively and incrementally, a prototype solution was created that automated anomaly detection and component localization. The system consists of five modules - Data Extraction Module, Data Pre-Processing Module, Detection Module, Localization Module, and Information Module, discussed in detail in Chapter 6.

We used a time-series deep learning model and unsupervised machine learning techniques overviewed in Chapter 2 to perform anomaly detection. In addition, we conducted experiments with the resultant system using a dataset from an industry-based microservices-based cloud system. We also experimented with the proposed detection method on a publicly available benchmark dataset.

1.5 Impact of the Proposed System

The proposed solution: (i) detects performance anomalies of a microservice through monitoring the performance metric data extracted from the tracing events using a novel approach of a prediction-based anomaly detection technique which combines a time-series model and unsupervised learning algorithms - LSCP, Isolation forest, and One-Class SVM, and (ii) locates the causal components for the detected anomalies. By using the proposed combination of LSTM and an unsupervised learning algorithm, there is no need to explicitly set a static threshold to score the anomalies. Instead, it uses a dynamic approach by making use of an unsupervised outlier detection technique such as LSCP or Isolation Forest, thus making the system entirely automated, unlike the static rule-based thresholding approaches which might require updates to the explicit threshold values when the data or load varies over time (discussed in Chapter 3 and Section 4.3).

The output of the proposed system is an anomaly report that consists of the time-interval of the occurrence of an anomaly and a directed acyclic graph of request flow highlighting its causal component.

In a microservices-based cloud environment, multiple instances of all the microservices need to be monitored at once. This requires checking the logs of multiple services and track one user request through multiple systems. The proposed system is capable to do and eases the monitoring process compared to the existing framework which might not be reliable. Also, in a microservices-based cloud system with numerous microservices and its multiple instances, the proposed system is helpful to locate the root cause of an anomaly. Thus, it overcomes the burden of SREs in manually analyzing and tracking the root-cause or faulty microservice/component.

1.6 Novelty

The novelty of the system lies in the detection module where the anomalies are detected by using an ensemble of a time-series deep learning model, LSTM, and unsupervised learning algorithms such as (i) LSCP, (ii) Isolation forest, and (iii) Once-Class SVM. The idea is to forecast the values of performance metrics (for example, response time) using the time-series forecasting model and then use an unsupervised learning algorithm to detect the anomalies. This avoids the burden of defining static thresholds used in the existing monitoring framework.

The results show that the two novel detection ensembles (“LSTM+LSCP” and “LSTM+Isolation forest”) perform better than the existing static error thresholding method, resulting in average F1-scores of 86% and 84%, average precision scores of 82% and 77%, and average recall scores of 91% and 93% respectively.

1.7 Thesis Contribution

The work involves several contributions which are presented in this section. The thesis contribution as a whole is on the system for anomaly detection and localization for a microservices-based cloud system and the main contribution is a novel ensemble technique used for detecting anomalies in the detection module of the proposed system. This thesis

sheds light on microservices, propagation of requests through microservices in the cloud system, provides insights on distributed tracing events of microservices. The work explores how the performance metric data could be extracted from trace events along with relevant features that could be used for the detection and localization process which is one of the main contributions. This thesis explains how neural networks such as LSTMs (Long Short-Term Memory) and various unsupervised outlier detection algorithms are used as general-purpose anomaly detectors. This work integrates the time series forecasting deep learning model with unsupervised outlier detection methods to create an ensemble anomaly detection system which is the main contribution. It involves the evaluation of 8 different integrated methods: four different ensembles of LSTM with LSCP, one hybrid of LSTM with Isolation Forest, LSTM with One-Class SVM, and two different ensembles of LSTM with Static thresholding methods across six different set of experiments. This novel ensemble was also tested on a public benchmark dataset (Numenta Anomaly Benchmark). The final contribution of the work is the identification of the root-cause for the detected anomalies.

1.8 Thesis structure

Chapter 2 describes the relevant background concepts. Chapter 3 reviews the literature and describes the research gap. Chapter 4 details the cloud system and provides an analysis of the problems. Chapter 5 provides insights into the research methodology that we followed while conducting this research. Chapter 6 gives a detailed design and description of the proposed system for anomaly detection and localization. The details of the implementation, experiments on the third-party dataset, and their results are discussed and a comparison with related work is demonstrated in Chapter 7. Chapter 8 describes the impact of the proposed system, discusses the challenges encountered and alternate analysis. It then summarizes the thesis work and lists some items for future work.

Chapter 2

2 Background

This section begins by providing insights on Microservices, followed by theoretical concepts of Anomaly Detection such as its challenges, types of anomalies, how machine learning could be used for anomaly detection and their evaluation metrics. The second half of this section is dedicated to the background on technical concepts required for supporting the practical implementation of the idea in this thesis. It starts by explaining the different types of unsupervised machine learning algorithms used in our work. Later, the section introduces the basic concepts of neural networks such as what a neural network is, how the learning of a network takes place and how a neural network could be used for sequential data using Recurrent Neural Network and Long Short-Term Memory (LSTM) network.

2.1 Microservices

Microservices architecture is a cloud-native architectural approach to develop software applications where a single application is composed of many loosely coupled and independently deployable smaller modules/components [8,10]. These independent modules or services have their own stack, data model, and databases. Each service provides its own business functionality and communicates with other services using pre-defined network APIs (Application Programming Interface) [9]. It is easier to develop applications, scale them faster with new features, and developers have the flexibility and autonomy to select the best breed of tools and programming languages to design and deploy each service. With the Microservices, as technology evolves, services can be replaced easily to reflect the efficient way to better their applications [8,9,10].

Applications built using microservices needs to be monitored to avoid possible future failure and outages. Without monitoring it is impossible to know if the services are provided without violating Service Level Agreements or not. Monitoring a microservices application is challenging unlike monitoring a monolith application. Every interaction between one service to another dependent service could be a potential point of failure.

Failure of one dependent service could result in upstream effects on the overall performance of the application [13].

2.2 Anomaly Detection

An anomaly is an unexpected, rare event or observation that is significantly deviating from the majority of normal data. For example, an abnormal behaviour of the system results in poor system performance due to high CPU consumption, high latency, etc. The process of identifying these unexpected events is termed as anomaly detection [1]. Anomaly detection is applicable in a variety of domains such as event detection in sensor networks, intrusion detection in cyberspace, in financial transactions, radio frequency transmissions, driving patterns, e-commerce, health care, etc. These anomalies are critical as they contain hidden significant information that is hard to find. For example, anomalous readings from different sensors could mean faulty road or weather conditions that could lead to road accidents or abnormal points from MRI images that could indicate the presence of malignant tumors.

2.2.1 Challenges in Anomaly Detection

Anomaly detection is a broad and hard problem. There are several challenges in the process of detecting anomalies. Some of the common challenges are listed as follows.

- Determining whether an event is normal or abnormal among a given set of events can be difficult, especially if the event-value lies close to the "edge" of normality and abnormality.
- Anomalies are masked. Certain anomalies arising out of fraudulent activities may appear normal thus making it difficult to define normality or normal behavior.
- Considering the dynamic nature of data, the anomalies detected today might not be anomalous in the future.
- An anomaly detection system that is effective for one dataset may not be effective for another dataset.
- Target labels indicate whether a data point is normal or not. The availability of labeled data, and thus the validation of detection models in the absence of label information is a huge problem [1].

2.2.2 Types of Anomalies

Anomalies are broadly classified into three main categories:

1. **Point Anomalies:** A single instance of data that deviates from the rest of the data points is considered as a point anomaly.
2. **Contextual Anomalies:** The abnormality is context-specific. An instance of data that is anomalous when viewed in a particular context but normal otherwise is considered as a contextual anomaly. For example, usage of air conditioning during winter is anomalous whereas, usage during summer is normal.
3. **Collective Anomalies:** A group of related data instances is anomalous with respect to the entire dataset, but not an individual instance. For example, the response time of a single request at the moment might be normal, but the sum of response times of requests for a 5-minute interval might result in anomalous system behavior.

These anomalies are inter-connected. A point anomaly could become contextual if a context is applied. Point anomalies could become collective anomalies if they are grouped together.

Apart from the type of anomalies, the detection process needs to consider the nature of input data [1]. The input data is a collection of data instances which could be events, observations, patterns, or vectors. Each observation is described by a set of attributes or features such as response time, CPU utilization, network usage of a system, etc. Input data can be univariate or multivariate depending upon the number of attributes. Univariate data consists of a single attribute, such as performance data with only response time values as its feature. Multivariate data consists of multiple attributes, such as performance data consisting of the total number of requests, duration, response code, timestamp, etc. Attributes are represented by continuous, binary, or categorical values. The selection of the desired detection techniques is dependent on the nature of the data. For example, in sequence data such as time-series or genome sequences, the data instances are linearly ordered. In spatial sequences such as traffic data, each data instance is related to its neighboring instance [1].

2.2.3 Machine Learning for Anomaly Detection

Machine learning (ML) is a subset of Artificial Intelligence that provides systems the ability to automatically learn from data without any human intervention, build models and improve from experience without being explicitly programmed to perform the task [2][3]. As given in Figure 2.1, a machine learning process creates a model, to make predictions or decisions, during the learning phase by training the chosen algorithm on training or sample data that has been pre-processed. The model is further evaluated by testing on new, unseen data.

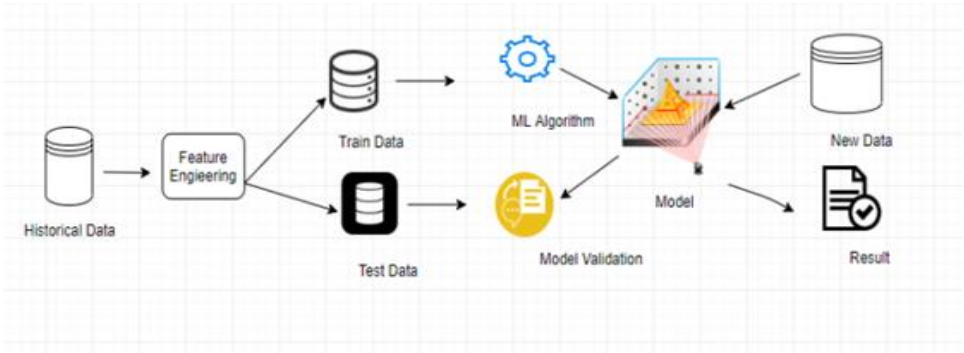


Figure 2.1: Machine Learning Process

The presence of target labels associated with each data instance is another aspect for deciding the anomaly detection technique [1]. Target labels provide an indication of whether a given data instance is normal or anomalous. Based on the availability of the labels, an anomaly detection system can use one of the 3 modes of machine learning techniques: Supervised, Unsupervised, and Semi-Supervised.

Supervised Learning requires a labeled training dataset that contains both normal and anomalous points in order to build the model to classify future data points. Unsupervised Learning does not require labeled training data and assumes that only a small percentage of data is anomalous and the anomaly is statistically different from the normal samples. It scores the data solely based on the natural features of the dataset. Distances or density scores are used for evaluation of what is normal and what is abnormal. Semi-Supervised learning falls between the above two categories. It uses a small amount of labeled data with

a large amount of unlabeled data for training the model and assumes that the training data consists of labels only for normal class [1-6].

2.2.4 Metrics to Evaluate Anomaly Detection

In Anomaly detection, we tend to have only two sets of targets or labels: normal and abnormal points. Hence, for such a two-class or binary classification, various techniques can be used to evaluate the ML models used for the detection process, provided the label information is available to assist the evaluation process.

2.2.4.1 Confusion Matrix

The commonly used approach for evaluation is by creating a confusion matrix, which is a table of two rows and two columns displaying the actual values and predicted values as shown in Table 2.1.

		Predicted Values	
		Normal Class	**Anomaly Class**
Actual Values	**Normal Class**	True Negatives (TN)	False Positives (FP)
	Anomaly Class	False Negatives (FN)	True Positives (TP)

Table 2.1 Confusion Matrix for two classes

- True Positives (TP) are the number of instances that are actually abnormal or anomalies (such as high response time of a system) and the model also predicts it as abnormal.
- True Negatives (TN) are the number of instances that are actually normal points and the model also predicts it as normal.
- False Positives (FP) are the instances that are predicted as abnormal when they are actually normal data points, giving false alarms.
- False Negatives (FN) are the number of instances that are predicted as normal by the models, but in reality, they are abnormal data points, indicating a miss during the detection process.

2.2.4.2 Accuracy

Accuracy measures how often the model predicts correctly. It is the proportion of correctly classified instances to the total number of instances as given in Equation 2.1. Accuracy is not a preferred metric when there is a severe class imbalance.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{2.1}$$

When the dataset class is highly imbalanced the following metrics are used for evaluation of the algorithms.

2.2.4.3 Precision

Precision is a measure of the accuracy of positive predictions or anomaly identification. It is defined as the proportion of data points that were correctly predicted as anomalies over the total number of data points that were predicted as either anomaly or normal as given in Equation 2.2.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

2.2.4.4 Recall

Recall also known as Sensitivity or True Positive Rate (TPR), measures the proportion of data points that were correctly predicted as anomalies over the total number of anomaly points present in the dataset as given in Equation 2.3.

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

2.2.4.5 F1 Score

F1 Score is the harmonic mean of precision and recall, as given in Equation 2.4, which gives a better measure of incorrectly predicted labels than the Accuracy metric. F1 score gives equal weight to both precision and recall measures.

$$F_1\ Score = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{2.4}$$

2.2.4.6 Specificity

Specificity, also known as True Negative Rate (TNR), measures the proportion of data points that were correctly predicted as normal points over the total number of normal points present in the dataset, as given in Equation 2.5.

$$Specificity = \frac{TN}{TN + FP} \tag{2.5}$$

2.3 Unsupervised Machine Learning Algorithms

This section provides an overview of the different unsupervised learning algorithms used in our proposed Anomaly Detection and Localization system. The section explains the following algorithms: K-Nearest Neighbor, Local Outlier Factor, K-Means Clustering, Isolation Forest, One-Class Support Vector Machine, Locally Selective Combination of Parallel Outlier Ensemble (LSCP).

2.3.1 K-Nearest Neighbor

K-Nearest Neighbor (KNN) is a density-based anomaly detection technique that assumes that the normal data exists around a dense neighborhood whereas abnormal data lies far away. For a given data point, KNN uses its distance to its *kth* nearest neighbor as the outlying score or anomaly score. Distance is used as a way to measure density. Different variants of detectors can be used such as:

a) Largest, which uses the largest value of distance, i.e., distance to *k*th neighbor as the outlying score.
b) Mean, which uses the average distance to all its k neighbors as the outlier score.
c) Median, which uses the median of all its distance values to its '*k*' neighbors as the outlier score.

Distance metrics such as Euclidean, Manhattan (or) Hamming distance can be used to compute the distance between data points. If the outlier score is high, then it can be considered as an anomaly.

2.3.2 Local Outlier Factor

Local Outlier Factor (LOF) is an unsupervised density-based algorithm that relies on k-nearest neighbors. The idea behind this algorithm for anomaly detection is that the density around an outlier will be significantly different than the density around its neighbors. LOF measures the local density deviation of a given data point with respect to its neighbors. The data points that have a substantially lower density than their neighbors are considered as anomalies. LOF tries to determine how isolated each data point is relative to other data points. LOF has 4 steps:

2.3.2.1 Steps in LOF

The local outlier factor algorithm involves 4 steps to detect the outliers in the dataset. The steps are as follows.

a) **Find K-Distance and K-Neighbors** – The first step is to choose a number ‘k’ of neighboring points and for a given point ‘p’, find the necessary radius ‘r’ to have ‘k’ points within a distance ‘r’ from ‘p’. The more isolated a point is, the farther it will have to search for its neighboring points, whereas for normal data points it doesn’t have to search too far to find its ‘k’ neighbors, as depicted by Figure 2.2. [98].

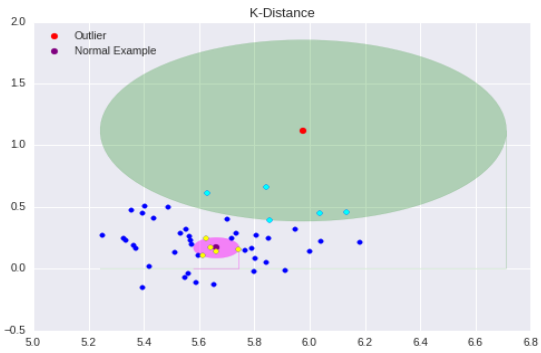


Figure 2.2: K-Distance for outlier (the red point) is larger than normal points (maroon point). Image adapted from [99].

b) **Compute Reachability Distance -** Reachability Distance determines which neighbors of a given point ‘p’ also consider that point ‘p’ as its neighbor as given

in Figure 2.3. For a point x and one of its neighbors y, reachability distance shows how a point y perceives the distance to x. Reachability-Distance for two points x, y can be defined as the maximum of the K-Distance of y and the distance between x and y. If point x is one of the y's k-nearest neighbors, then distance(x,y) will be less than K-Distance(y). Hence, it makes reachability Distance(x,y) = K-Distance(y). If point x is not one of y's k-nearest neighbors, then distance(x,y) will be greater than K-Distance(y). Hence, it makes Reachability Distance = Distance(x,y).

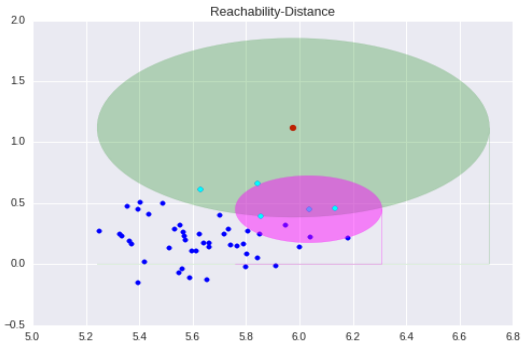


Figure 2.3: Reachability Distance determines which neighbors of a given point 'p' also consider that point 'p' as its neighbor. The outlier (red point) is not contained in K-neighbourhood by its neighbor (aqua-colored dots). Image Adapted from [99].

c) **Determine Local Reachability Distance (LRD)** - LRD provides a statistical density for each point. For a point x, LRD is equivalent to the inverse of average reachability distance of x's neighbors. LRD tells how far it needs to travel from a given point to reach the next point or cluster of points. The lower the LRD is, the less dense it is and hence longer it needs to travel.

d) **Local Outlier Factor (LOF)** - Each point's LRD is compared to its neighbors' LRD to compute the LOF for each point. LOF is the average ratio of LRDs of neighbors of point x to LRD of point x. For most points values of LOF should be close to 1. If LOF >> 1, it indicates that the density of point x is low compared to its neighbors and hence, it has to travel longer from point x to reach next point/cluster of points (or) indicates it is far from dense areas. Therefore, the higher the LOF, the more it is likely to be an outlier [98, 99].

2.3.3 Isolation Forest

Isolation Forest (IF) is an unsupervised learning technique that uses the concept of isolation instead of distance or density measures [100]. The basic assumption made by this algorithm is that anomalies are very and different. The algorithm tries to separate each data point in the dataset. The intuition here is that an anomaly can be easily separated in a few steps while the normal points which closer could take more steps to be separated.

2.3.3.1 Steps in Isolation Forest

The detection process of an Isolation Forest involves 4 steps. Each step has been explained as given.

a) Sampling - The dataset is sampled for training the model, as given in Figure 2.4.a). The sample proportion can be different depending upon the presence of noisy data in the underlying dataset.

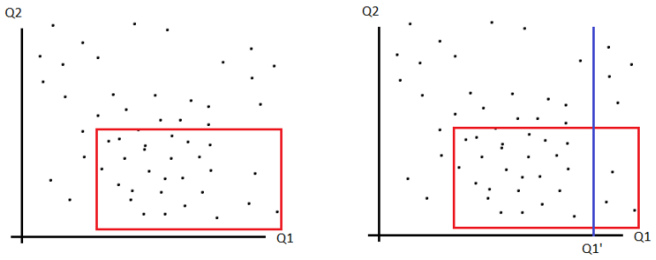


Figure 2.4: a) Sampling (left). b) A Split value selected to form a tree (right). Image Adapted from [101].

b) Binary Decision Tree - For the sample drawn from step a), a binary decision tree is built. This step randomly selects a feature and then selects a split value between the maximum and minimum values of the selected features, as given in Figure 2.4.b).

c) Create Forest - The two sub-data set formed by a binary split in previous step b) is further split to form a tree. Basically, step b) is repeated iteratively to create a collection of trees, a forest. Fewer and different data points are segregated quicker i.e., it takes less path for them to be isolated, as given in Figure 2.5. To isolate a

sample, the number of splitting required is equivalent to the path length from the root node to the leaf node.

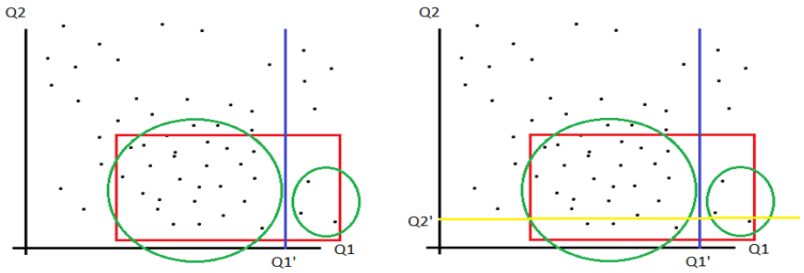


Figure 2.5: Data points at the lower right corner are easier to isolate (right). Image Adapted from [101].

d) Calculate Anomaly Score - Each data point is fed into the trained forest for each tree. Anomaly score is calculated for each tree and average is taken across different trees to obtain the final anomaly score for an entire forest for a given data point.

2.3.4 One-Class SVM

One-Class SVM (OC-SVM) is an unsupervised learning technique based on the working of the Support Vector Machine (SVM). SVM is a supervised classifier that tries to find an optimal hyperplane having a maximum margin in order to separate two classes of data points [102], as given in Figure 2.6. The idea of SVM for anomaly detection is to learn a decision function that is negative for regions with a small density of points and positive for high dense regions.

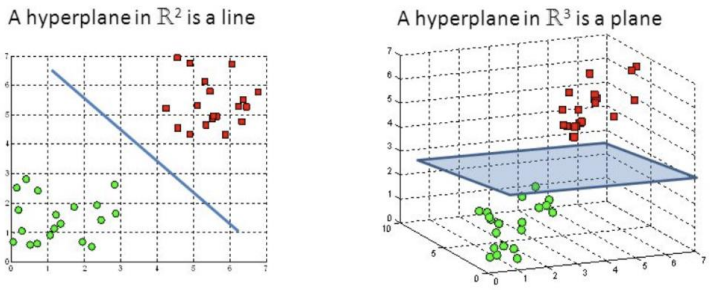


Figure 2.6: Hyperplanes for linearly separable data and non-linear data

One Class SVM separates all the data points from the origin and maximizes the distance of hyperplane from the origin which results in a binary function returning positive and negative values for regions on either side of the hyperplane. The model uses hyperparameters such as ‘nu’, known as outlier fraction, which is the proportions of outliers expected in the data, kernel type such as ‘rbf’, which enables SVM to project the non-linear input into a high dimensional feature space, ‘gamma’ is a kernel co-efficient controlling the influence of training samples [103]. The model returns –1 for an outlier and +1 for a normal data point.

2.3.5 Locally Selective Combination of Parallel Outlier Ensembles (LSCP)

Locally Selective Combination of Parallel Outlier Ensembles (LSCP) is a framework used for the detection of outliers using an ensemble of unsupervised outlier ensembles [91]. The LSCP is a framework available in Python Outlier Detection (PyOD) toolkit. PyOD is an open-source, scalable Python toolkit for detecting outliers.

2.3.5.1 Ensemble Methods and Types

Ensemble methods use multiple learning algorithms to obtain a better performance that could not be achieved by using individual models. In unsupervised outlier ensembles, it is quite challenging to build a combination of detectors without labels or ground truth. There are 2 variants of ensembles - parallel and sequential. In sequential, the base detectors/learners are generated sequentially where the dependence between the learners are exploited. In parallel, the base detectors are generated in parallel exploiting the independence between the detectors [92].

2.3.5.2 Advantages of LCSP

Existing parallel outlier ensembles combine all the base learners without considerable selection, limiting the combination benefits since individual detectors may not be capable of identifying all the outliers [93]. The performance of ensemble through good detectors can be reduced or nullified by the presence of bad detectors while averaging the results. Also, the detectors consider all the training points in the dataset to determine the outlier

exploring global data relationships instead of considering the local regions in the dataset. LSCP overcomes these limitations by emphasizing on data locality [93].

LSCP is motivated by the principle of Dynamic Classifier System (DCS) [93], a supervised ensemble framework, that the base classifiers are likely to specialize in local regions than identify outliers from all the unknown test instances [91]. Based on this idea, LSCP defines a local region for each test instance, then identifies the competent base detector(s) in this local region which in turn generates an outlier score for the test instances. LSCP explores global data relationships by training the detectors on the entire data and explores local data relationships by emphasizing locality during detection combination. LSCP is compatible with diverse types of base detectors.

2.3.5.3 Steps in LSCP algorithm

1. **Base Detector Generation** - The base detectors used in LSCP can be heterogeneous or homogeneous. For homogeneous base detectors, different hyper-parameters initialization & subsampling of training dataset can be done to introduce diversity so that the model can learn distinct characteristics of data. For a given training and testing data, the algorithms generate a set of base detectors with different values of hyper-parameters, say a set of KNN or LOF detectors with distinct *neighbors* or *Minpts*. The training set is used to train all the base detectors, which are further tested on the same training data resulting in an outlier score matrix **O(X train)** with score vectors from all the base detectors which are normalized using Z-normalization [91].
2. **Pseudo Ground Truth Generation** - Since this is an unsupervised technique with no labels or ground truth, two methods are used to generate pseudo truth with Outlier matrix score **O(X train)** which can be used by LSCP to evaluate the competency of the detector. One is average scores of base detectors (LSCP_A) and the other is the maximum of all base detector scores (LSCP_M). This pseudo ground truth is generated using training data and used for detector selection alone.
3. **Local region Definition** – A set of nearest neighbors for each test instance is identified to determine that its local region. Firstly, a new feature space is constructed using randomly chosen t groups of [d/2,d] features. Then, using

Euclidean distance, ‘k’ nearest training objects to test instances in each group are identified. The training objects appearing more than t/2 times define the local region for a given test instance.

4. **Model Selection and Combination** – For each test instance, local pseudo ground truth is determined by fetching values associated with the local region from the target in step 2. Local training outlier score is obtained by retrieving outlier scores associated with training data in the local region, i.e., outlier score matrix from step 1. To evaluate base detector competency in the local region, LSCP uses the Pearson correlation to measure the similarity between local pseudo ground truth and the local detector score. Detector with high similarity is considered as the most competent local detector and its score is considered as the final score for that test instance.
5. **Dynamic Outlier Ensemble Selection** – If only one detector is most similar to the pseudo ground truth, then it would be risky to select only one detector for the unsupervised problem which can be avoided by selecting a group of detectors. There are two variations of LSCP ensemble – Maximum of Average (LSCP_MOA) and Average of Maximum (LSCP_AOM). In both cases, a group of detectors is selected in the local region of a test instance. LSCP_MOA takes the maximum of detector’s predictions as the outlier score whereas LSCP_AOM takes the average of detector’s predictions as the outlier score when pseudo target from step 2 is calculated using LSCP_M [91].

2.4 Neural Network

A neural network (NN), also known as Artificial Neural Network (ANN), comprises of simple computational units called nodes or neurons which use a mathematical or computational model for processing information [75]. The working principle of a neural network is inspired by the functioning of a human brain [76]. A neural network consists of several layers where each layer is made of neurons that are connected through a link. Each link is associated with a set of co-efficient or weights. A node receives input along with the incoming links which are combined with their respective associated weights as depicted in Figure 2.7.

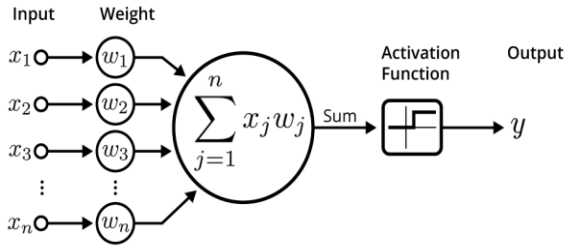


Figure 2.7: Functioning of a neuron. A simple network consists of one output neuron (Perceptron Model) where inputs are directly fed to the neuron. Image adapted from [79].

The products of inputs and weights from all the nodes across a layer are summed and further passed to an activation function to generate output signals [76-78]. Activation function, also known as a transfer function, acts as a gate determining to what extent an output signal should progress further in the next layer [77]. It maps input nodes to output nodes based on mathematical operations [80]. There are 3 types of activation functions, namely, Linear, Binary, and Non-Linear. Most of the neural networks use non-linear transfer functions which helps the model to create a complex mapping between input and output when the data is non-linear. Typical non-linear activation functions include: Logistic/Sigmoid, Tanh, Rectified Linear Unit (ReLU), Parametric ReLU, Leaky ReLU, Softmax, Swish [81].

2.4.1 Feedforward Neural Network

There are various classes of neural networks: Feedforward, Convolutional, Recurrent, Autoencoder, Generative Adversarial, etc. In this section, we describe a simple network, FFNN, to explain its working mechanism before we learn about LSTM.

A feedforward neural network (FFNN) is an artificial neural network where information flows only in one direction, from input nodes to output nodes without any feedback loops. Depending on the number of hidden layers, they can be classified into Single-layer perceptron or Multi-layer perceptron [82]. A single layer perceptron does not contain any hidden layers and consists of a single layer of output node(s), as depicted in Figure 2.7. A multi-layer perceptron consists of input, hidden, and output layers as shown in Figure 2.8.

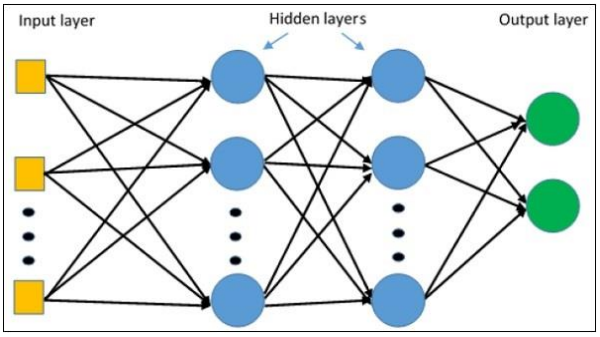


Figure 2.8: Multi-layer Perceptron

Input Layer - is the first layer of a network, also known as a visible layer since it is the exposed part of the network in which each node takes one input value from the dataset. The input layer is responsible for passing the input to the next layer, hidden layer, and does not perform any operation.

Hidden Layer(s) - are composed of most of the neurons in the network which are responsible for manipulating the incoming data with the weights and biases.

Output Layer - generates the result for the data passed through the network. It might consist of a single node or multiple nodes depending on the number of resultants objects expected to be returned.

2.4.2 Learning Process of Neural Network

The learning or training of a neural network is an iterative process involving the flow of information forward and backward across the layers of neurons. The first phase is forward propagation where the network receives input data, which is further passed to the next layer where it is transformed and fed to the next subsequent layer. After the data propagates across all the hidden layers, it reaches the last layer which generates the output. This network generated output is compared with the actual output to determine the error or loss using a loss function (such as Mean-Squared Error), to measure how accurate the result is.

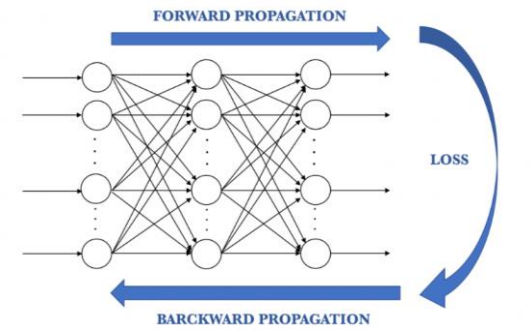


Figure 2.9: Training of Neural Network using Error back-propagation

The loss information is propagated backward from the output layer to other layers in the network one by one. The neurons of the hidden layers receive a significant fraction of the total loss based on their contribution to the output. The process is repeated until all the neurons have received the loss information [83]. Later, the weights are adjusted in such a way that the error is minimized. This is called optimization which aims at minimizing the loss function by tuning the parameters of the network such as the weights and biases. The technique used for optimization is called gradient descent which changes the weights in small increments by calculating the gradients of the loss function indicating the desired direction to reach minima. Gradient(derivative) is a measure of the change in loss value corresponding to a small change in network parameters [76]. A hyper-parameter called learning rate controls the size of steps or the amount of the weight adjusted with respect to the loss gradient, as given in Equation 2.6. This process is done for several epochs or iterations over the training dataset. The parameters move closer to their optimal values with every epoch. A smaller value of learning rate requires more iterations whereas larger learning rates require few epochs.

$$\theta = \theta - \gamma \frac{\partial L(\theta)}{\partial \theta} \tag{2.6}$$

(New weight = Old weight – Learning rate * Gradient)

Gradient descent variants Depending upon the amount of data used to compute the gradient of the loss function, there exist different variants of gradient descent such as batch, mini-batch, stochastic gradient descent, RMSProp, ADAGRAD, ADAM, etc.

2.4.3 Limitations of Basic Neural Networks for Sequential Problems

Vanilla or basic neural network takes in the input of fixed size which limits its usability when it comes to sequential data or a ‘series’ input which has no predetermined size. To overcome the fixed size issue, multiple sets of vanilla networks could be used, but a single instance from a ‘series’ input has a certain relationship with its neighboring instances and basic neural networks cannot explore such relationships between consecutive instances of a series input [84]. Most of the NNs assume that the data samples are independent of each other. Such assumptions do not hold true for data like speech, video, stock market data, language, etc., that exhibit temporal dependency. One such mechanism to account for sequence data is to use Recurrent Neural Networks (RNNs).

2.4.4 Recurrent Neural Network

RNNs are a generalization of feed-forward neural networks (FFNNs) with internal memory. It is referred to as recurrent since it performs the same operation for every input where the output of current input depends on the past output. RNN adds a looping mechanism to FFNN as depicted in the below Figure 2.10, which allows the flow of information from one step to the next step. This information is known as the hidden state (memory), which represents the previous inputs.

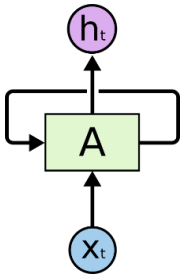


Figure 2.10: Recurrent Neural Network with Looping mechanism

Figure 2.11 depicts how RNNs can be used for modeling sequential data. As given in the diagram, there are ‘t’ instances or samples of data. The network takes in input X(0) to generate output h(0), which is fed along with the next input X(1) to generate second output h(1). Here, the output h(1) depends on current input, X(1), and its previous output, h(0). Similarly, this step is continued further until it generates ‘t’ outputs. This architecture of

RNN helps it to uncover the dependencies of input samples with each other and remembers the context behind the sequences, for example in language translation, while training.

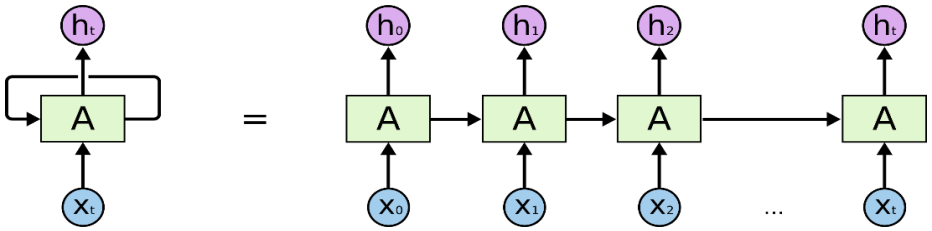


Figure 2.11: Unrolled Recurrent Neural Network. Image Adapted from [85]

2.4.5 Limitations of RNN

RNN suffers from short-term memory, meaning they are capable of handling short-term dependencies between the input samples. For example, in language modeling, it is easier to predict the next to last word in a sentence "Vehicles stop when traffic signal lights turn *red*" based on the previous ones, because the time gap between the relevant input and the place where it is needed to generate output is less. Here, in this example, RNNs does not have to understand the context or need not remember information from the previous sentences. But if the relevant input and the place where it is needed is separated by irrelevant data in between, then the RNN fails. For instance, "My name is XYZ and I was born in India...My passion is... I'm fluent in many *Indian* languages", the word 'fluent' indicates the next word to be 'language', but to predict which specific language(s), the RNN model needs to remember the context of the information from the relevant sentence that has been mentioned long ago. RNNs cannot handle such long-term dependencies. This is due to the problem of vanishing gradients [86].

Vanishing Gradient Problem:

As mentioned in Section 2.4.2, the gradient descent algorithm tries to find the global minima of the loss/cost function. The information travels from input to the output layer and the error is calculated and back-propagated back to the starting layer. The training is similar for RNN except that the information travels through time where the output from the previous time step is used as input for the next time step and error or cost function is calculated at every time step.

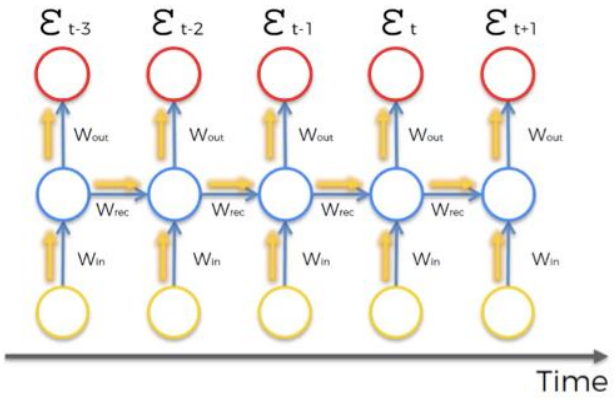


Figure 2.12: Training in RNN through back-propagation. Image adapted from [87].

For example, as depicted in Figure 2.12, at a given time step ‘t’, if weights are required to be updated, then the cost function/error term, E(t), needs to be backpropagated through the network. The weights of every neuron that participated in producing the output associated with this cost function are updated. In RNNs, not only the neurons present below the output layer at time step ‘t’, but all the neurons far back in time (say t-1, t-2, t-3) contributes to the desired output at time step ‘t’. Hence, the error is propagated through the network back through all the time steps [87].

For instance, to get from x-3 to x-2, x-3 is multiplied by Wrec (Weight Recurring), and to get to x-1 from x-2, x-2 is multiplied by Wrec to get to x-1 from x-2. The inputs are multiplied by the same value of weights many times. When a value is multiplied by a smaller weight, the product result gets reduced quickly. In NNs, weights are generally initialized with random values closer to 0 which later gets updated during the training process. Hence, if Wrec value is less and when the error is backpropagated, this Wrec will be multiplied with x, x-1, x-2, x-3, and so on, causing the gradient to decay to 0, which we call it as vanishing gradient problem. Similarly, if Wrec is high, it causes the gradients to become too large resulting in exploding gradient problem [87]. As a result, RNNs can remember things only for a short duration of time and tends to forget information over time. To counter the problem of vanishing gradient, Long-Short Term Memory (LSTM) architecture was introduced.

2.4.6 Long-Short Term Memory

LSTM is capable of handling long-term dependencies between the input samples. It can selectively forget or remember information and add new information without entirely modifying the existing information, unlike RNNs. An LSTM network comprises of memory blocks called cells; the green boxes as shown in Figure 2.13. There are four neural network layers (yellow boxes) inside every cell.

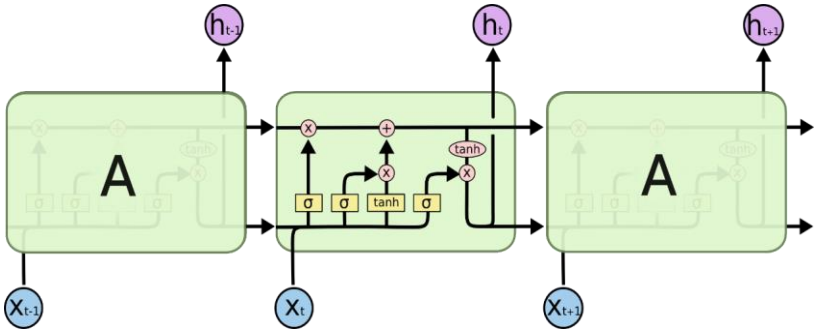


Figure 2.13: An LSTM network. Image adapted from [86].

Each cell transfers 2 states to the next cell: the cell state and the hidden state. These cells are responsible for remembering information that is important and this information can be manipulated through gating mechanisms. LSTM consists of 3 gates – Forget Gate, Input Gate, and Output Gate.

2.4.6.1 Forget Gate

Forget gate is responsible for discarding information that is no longer required (or) of less importance by the LSTM.

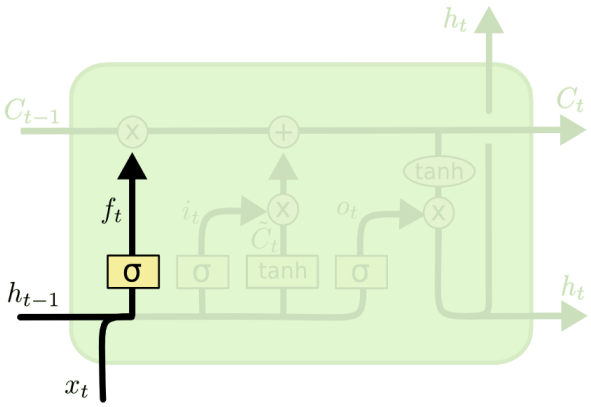


Figure 2.14: Forget Gate of LSTM. Image adapted from [86].

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right) \tag{2.7}$$

As shown in Figure 2.14, the gate takes in two inputs - $x_t$and $h_{(t-1)}$. x_t is the input at the current time step and $h_{(t-1)}$is the hidden state/output from the previous cell. The input is multiplied by the weight matrices and a bias is added, followed by a sigmoid activation function, $\sigma()$. Sigmoid activation function, $\sigma()$, decides which values to remove and which to keep. If it outputs '0' for a particular value in the cell state, the forget gate will tend to forget or discard the information. If the sigmoid outputs '1', then the forget gate will remember that information. This output, f_t, from the sigmoidal function is multiplied with the cell state, $C_{(t-1)}$ from the previous cell [86].

2.4.6.2 Input Gate

The input gate is responsible for adding new information to the cell state. It involves three steps for the addition of new information.

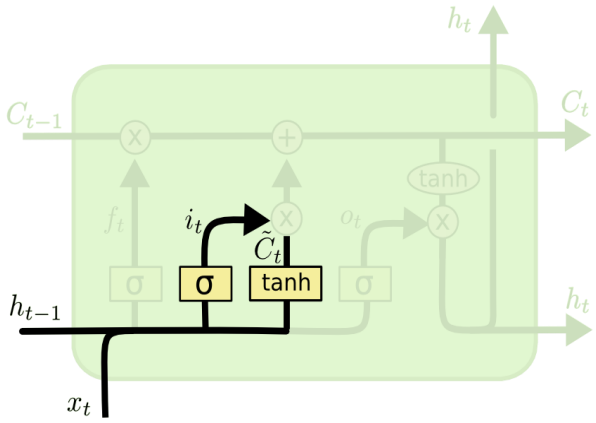


Figure 2.15: Input Gate of LSTM. Image adapted from [86].

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{2.8}$$

$$\tilde{C}_t = tan\, h(W_i \cdot [h_{t-1}, x_t] + b_c) \tag{2.9}$$

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t \tag{2.10}$$

1. The first step involves using a sigmoid layer, $\sigma()$, which acts as a filter in deciding which information to keep and update from x_t and $h_{(t-1)}$, as given in Figure 2.15 and described by Equation 2.8.
2. Next, a tanh layer creates a vector of all possible values from x_t and $h_{(t-1)}$that can be added to the cell state.
3. The third step would be multiplying the regulatory output, i_t, from step 1 with the vector values, $\tilde{C}_t$, from step 2 and later adding this to the previous cell state, C_{t-1}.

These steps make sure that only important information is added to the cell state and redundant ones [86, 88].

2.4.6.3 Output Gate

The output gate sends the filtered useful information from the current cell state to the next cell. This also involves 3 steps.

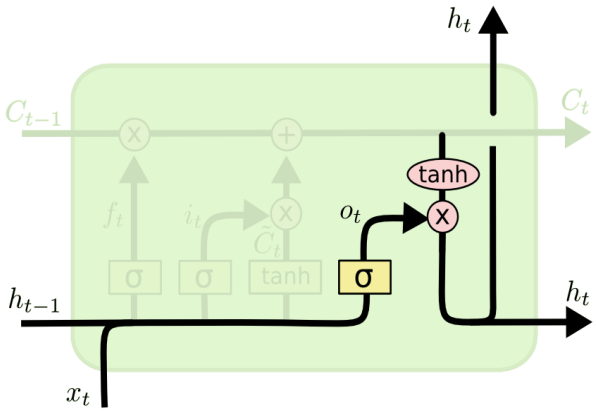


Figure 2.16: Output Gate of LSTM. Image adapted from [86].

$$o_t = \sigma\left(W_o \cdot [h_{t-1}, x_t] + b_o\right) \tag{2.11}$$

$$h_t = o_t * tan\, h\left(\tilde{C}_t\right) \tag{2.12}$$

1. The cell state, $\tilde{C}_t$, from the input gate is put through a tanh layer, as shown in Figure 2.16, to create a vector of values ranging from –1 and 1.
2. Next, a sigmoid layer, $\sigma()$, uses inputs $x_t$and $h_{(t-1)}$to filter the values that are going to output from the vector created in step 1, mathematically expressed as given in Equation 2.11.
3. We finally multiply the filter output from step 2, o_t, with vector values from step 1 and send it as the hidden state and output for the next cell, mathematically expressed as given in Equation 2.12 [86, 88].

The current cell state or the result from the output gate of the current cell is fed as input to the next cell, where the gates in the next cell repeat the same set of operations as discussed above until the last cell has reached in the network layer.

Chapter 3

3 Literature Review

In this section, we provide a literature review of anomaly detection techniques carried out in a distributed environment or computing systems in general, broken into main sections: Traditional Statistical approaches and Machine Learning-Based approaches. Under the ML approach, we discuss the state-of-the-art anomaly detection techniques used for time-series data by supervised, unsupervised, and deep learning techniques. Later, we provide information about the research gap identified by analyzing the existing approach.

3.1 Anomaly detection in a cloud system or general computing domain

Anomaly detection has been extensively studied and is an important topic in various domains such as health care, e-commerce, finance, cyberspace, astronomy, ecology, etc., [21-28]. Various detection strategies have been proposed in the literature specific to the computing domain, such as signature-based, observational, knowledge-driven, and detection methods such as statistical, machine learning, etc., [29]. Several existing research projects have addressed the specific problem of performance anomaly detection in distributed and computing systems in general which is discussed in this chapter later.

3.1.1 Traditional Statistical Detection Approaches

Beyond rule-based systems [38, 39] that require upper/lower bounds for performance metrics, researchers have exploited various statistical methods such as Markov model [30-32], correlation analysis [35-37, 42], regression analysis [33,34], gaussian-based techniques have been assessed to capture deviations in the system performance metrics. Bikash et al., in [36] identify variations in performance metrics in a cluster of Virtual Machines (VMs) using correlation analysis. The authors also characterize an anomaly by defining an anomaly signature for the pairwise correlations of CPU utilization of the VMs. Similarly, the correlation between system metrics, latency, and aggregated workload is analyzed and considered as a basis to uncover anomalies by Joao et. al in [35], Bikash et

al. in [36], and Manjula et. al in [37]. Yang et al., in [34] and Kang et al., in [33] propose a regression-based diagnostic framework to model the relationship between system performance and application metrics to detect performance anomalies and determine its root-cause.

Gu et al., in [32] and Tan et al., in [31] describe a system based on Markov chain model to capture the changing patterns of different measurement metrics to predict metric values for next *k* time units and further feed to a Bayesian classifier to determine anomaly symptoms. Samir et al., in [60] designed a Detection and Localization system for Anomalies (DLA) that monitors and analyzes performance-related anomalies in container-based microservice architectures. They adopted Hierarchical Hidden Markov Models (HHMM) and Correlation Analysis to model the relation between the monitored metrics of the container, node and service, and the variation in response time under different load scenarios.

3.1.2 Machine Learning-Based Detection Approaches

Due to the scale and complex nature of cloud or distributed environment, there has been a tremendous shift in the adoption of machine learning techniques. Machine learning approaches are divided into 3 categories, namely supervised, semi-supervised, and unsupervised, depending upon the level of supervision required by the models. We also discuss the state-of-the-art approaches under each category. This section also provides a review of existing literature on deep learning techniques for anomaly detection.

3.1.2.1 Supervised Learning Approach

Daniel et al. [43] describe an approach that uses metric data and log information to select different classifiers trained via Support Vector Machines using monitoring metrics. The technique aims to find anomalies over every time intervals called windows by using moving average and entropy of metrics data in each window as additional features for training the classifiers.

In [46], Sauvanaud et al. propose an Anomaly Detection System (ADS) consisting of 3 modules, designed to detect anomalies by learning the behaviour of services in VMs using supervised machine learning techniques such as Random Forests, Neural Networks,

Nearest Neighbors, and Naive Bayes on CPU, Memory, Disk, and Network performance data.

Qingfeng et al. [45] designed an Anomaly detection System (ADS) to detect and diagnose anomalies in a container-based microservices. The proposed system consists of 3 modules: Monitoring module, that collects performance metric data of containers, Data Processing Module analyzes data and detects anomalies using SVM, Naïve Bayes, Nearest Neighbors, and Random forests algorithms and Fault Injection Module validates the model under different system fault conditions. Further, once an anomalous metric in service is detected, the time-series data of all the containers running that service is analyzed by the DTW algorithm to measure the similarity between the time-series performance data of the given containers. And the most anomalous container which has the maximal distance from the others is found.

3.1.2.2 Unsupervised Learning and Deep Learning Techniques

The most widely used unsupervised anomaly detection technique for point anomalies is K-Nearest Neighbors (KNN), which calculates an anomaly score based on the distance to 'k' nearest neighbors for a given data point used by Bikash et al. in [36], James et. al in [58], and Kanishka et. Al in [59]. Breunig et al., in [55] proposed a popular unsupervised method for local density-based anomaly detection known as Local Outlier Factor (LOF) where the k-nearest-neighbors set is determined for each instance by computing the distances to all other instances. LOF has been used for anomaly detection in cloud applications [53], for workload patterns [54], network intrusions [57].

Xiao et al., in [61] developed TaskInsight that detects performance anomalies in cloud applications using clustering algorithms by analyzing the system-level metrics, such as CPU and memory utilization. The anomaly score of an instance is the distance to the next large cluster. The problem of choosing the right number of clusters arises in the clustering approach [56].

ADVec algorithm developed by Twitter (Vallis et al. [48]) based on the generalized Extreme Studentized Deviate (ESD) test, combined with robust statistical approaches and piecewise approximation. The technique uses statistical metrics such as median, and

median absolute deviation (MAD), and piecewise approximation of the underlying long-term trend to detect anomalies in the long-term time-series data. It uses the piecewise method to approximate the underlying trend in a long-term time series. The trend is computed as a piecewise combination of short-term medians. The length of the windows in the piecewise approach is chosen such that the windows encompass at least 2 periods of any larger seasonality.

Imam et al. [44] use the Microsoft ML time series algorithm on application log files to forecast the performance, detect anomalies, and analyze to determine the application or source that caused it. The ML time series algorithms include 2 algorithms : (i) The ARTXP algorithm, which is optimized for forecasting the next probable value in a series, (ii) The ARIMA algorithm, to improve the accuracy of the long-term forecasting. After forecasting, the proposed approach introduces the concept of anomaly index, computed by looking for a maximum value across a time-period and comparing it to the average. The technique observes a change in anomaly index, its trend, and alerts when the value of the anomaly index is large.

Sasho et al. [47] address the problem of anomaly detection in a large-scale distributed environment by proposing an unsupervised response time anomaly detection of Variational autoencoders and dynamic error thresholding. Variational autoencoders forecast the metric values from distributed tracing records, the forecast errors are modeled as gaussian distribution. The validation set is being used for threshold setting where for each window per sample in validation set the trained model is applied. Errors between reconstructed and an observed window of events not within the high-level of the confidence interval of Gaussian distribution is considered as an anomaly. The probability of new test data within a high-level of confidence of the Gaussian distribution confidence interval is computed and outputs are kept in a queue of size the same as (tolerance) for each new window. The tolerance module checks whether the average probability of all the points in the queue is greater than the error threshold. If this is the case, the submodule flags this part of the time series as unstable and reports an anomaly.

Similarly, Haowen et al. [50] propose an unsupervised anomaly detection algorithm, Donut, based on dimensionality reduction and generative model, Variational auto-encoder (VAE) for seasonal Key Performance Indicators (KPIs) for web applications (e.g., page views, number of orders, online users) with local variations. The technique uses an optimal threshold value to determine if a data point is an anomaly or not.

Mohsin et. Al in [56] developed DeepAnt, an unsupervised anomaly detection technique, which consists of two modules – time-series predictor and anomaly detector. The predictor module uses on Convolutional Neural Network (CNN) to forecast time-series data. The actual and forecast value is passed to the detector, which uses Euclidean distance to measure the discrepancy and detect the anomalies if the distance measure exceeds beyond the threshold set.

Malhotra et al. [49] proposed a model of stacked Long-Short Term Memory (LSTM) networks to enable learning of higher-level temporal features to detect anomalies in time series data. The network was trained on normal data and used as a forecaster over a number of time steps. The forecast errors were modeled as multivariate gaussian distribution, which was used to assess the likelihood of anomalous behaviour. Validation sets were used to determine the anomaly cut-off. If the likelihood or the probability of a test point is greater than the threshold then the point is considered as an anomaly.

Chen et al. [51] designed a framework to integrate unsupervised anomaly detection and trend prediction altogether. The framework, SeqVL - Sequential VAE LSTM, combines variational auto-encoders and LSTMs, where VAE is used for unsupervised anomaly detection and LSTM is used for trend prediction. The detector (VAE) boosts its performance by training the model with segments in sequential order that is maintained by the predictor (LSTM). The re-encoded time series output from the VAE block is fed to LSTM to make robust trend predictions. The squared error of encoder from decoder segments is checked against a threshold to detect anomalies.

Leandro et al. [52] extended eBay's Atlas algorithm to automatically detect anomalies in unlabeled seasonal time series data. The proposed algorithm, MULDER uses a 'surprise' metric from the time series, which is then statistically analyzed to determine anomalies. A

percentile window is applied to the surprise metric (computed by differencing actual and expected value) and calculates the 10^{th} and 90^{th} percentile over the subset and later performs a 3-standard deviation test within a window of length, n, sliding across the two percentile time-series.

3.2 Analysis and Research Gap

The papers discussed above were analyzed to identify the research gaps. The analysis of the aforementioned existing works is described as follows.

Several existing works have exploited various statistical methods such as Markov model [30-32], correlation analysis [35-37, 42], regression analysis [33,34], etc. Gu et al., in [32] and Tan et al., in [31] describe a system based on the Markov chain model along with the Bayesian network whereas Samir et al., in [60] use HMM with correlation analysis to perform detection of anomalies. Hidden Markov Models perform better only if their assumptions hold true such as (i) state transitions depend only on the current state, not on anything in the past, and (ii) The total number of states is pre-defined. This might not be true when the performance metric data is sequential time-series data, where the future values depend on the past and the number of states cannot be fixed in sequential data which is dynamic, decreasing the performance of Markov models.

Many statistical detection techniques often assume that the distribution and density of data are known apriori or can be inferred to detect well-known anomalies. They tend to exhibit sensitivity in case of load variations when these assumptions do not hold true [40]. For non-linear time-series data, to build a statistical method that can describe this data, we might need to build a piece-wise function. Hence, statistical correlation methods are expensive to learn and require a large volume of training data for non-linear correlations [41].

Qingfeng et al. in [45], Sauvanaud et al. in [46] and Daniel et al. [43] use supervised algorithms such as SVM, Naïve Bayes, Nearest Neighbors, and Random forests to classify normal and abnormal data, i.e., find anomalous data/service from the normal data. The detection algorithms are trained by the data containing label information on whether a point

is an anomaly or not. For practical usage, (i) it is inefficient to label data involving time-series and concept-shift, given the dynamic nature of data, as the method might fail to detect when it encounters a new or unknown anomaly (ii) expensive to label data requiring manual efforts and (iii) time-consuming because the real-world data is voluminous. The need for labeling might increase when the volume and complexity of the data increases.

Bikash et al. [36], James et. al [58] and Kanishka et. al[59] use the unsupervised algorithm, KNN, for detecting anomalies which is highly dependent on the value of 'k' neighbors and might fail if there aren't enough neighbors, and it is computationally expensive. LOF has been used in cloud applications by Tian et al. [53], for workload patterns by Tao et al. [54], network intrusions by Lazarevic et al. [57]. It assumes that the neighbors of the data instances are distributed in a spherical manner, which has limitations when the data tends to have a linear distribution, i.e., if the normal data points are distributed in a linear way [56]. However, it doesn't fit well for detection purposes in a service-oriented system [53], where there are large datasets and high dimensional data with multiple features. Also, the aforementioned technique fails to capture the temporal dependency between the data points in sequential time-series data.

In [48], Vallis et al. developed the ADVec algorithm, based on the ESD test combined with statistical methods, which has limitations on a large distributed environment based on service-oriented and microservice architectures. If the time-series exhibit more than two different normal behaviours or operations, similar to the time-series generated by the microservices' system, then the algorithm might not be able to learn this normal information [47]. Secondly, this approach uses three parameters out of which one is used as the anomaly threshold, describing the level of statistical significance to accept or reject anomalies.

In [44] Imam et al., this approach indirectly uses thresholding rule while computing anomaly index. Firstly, it is difficult to determine which time-period needs to be used to compute anomaly index, given the dynamic nature of data and secondly, it states larger the anomaly index, stronger are the signs of an anomaly. The question here is how to determine the large value of the index. If the largest anomaly index is determined by sliding across

time-periods, then again it is uncertain with respect to choosing time-period/sliding window interval as mentioned earlier.

In [47], Sasho et al. use Variational autoencoders with probability-based dynamic error thresholding, where this approach uses window size and confidence interval as the basis for thresholding. The tolerance module checks the average probability of all points in each window exceeds the error threshold or not. If it exceeds then it declares the entire window or time-series as an anomaly. The approach concludes the entire time-series if a single data point results in a larger value, whereas the other individual points might not be anomalous. The approach seems to not detect point anomalies. Secondly, tolerance or window size might never be the same in practical usage and might require an update with the new incoming stream of data over time.

Malhotra et al., in [49], Mohsen et al., in [56], Haowen et al. in [50], and Chen et al., in [51] proposed different novel anomaly detection methods for time-series data respectively. These methods use a static threshold to score an anomaly which might require an update on a timely basis with a new stream of data, because, in a distributed environment such as a service-oriented and microservices system, the data can exhibit more than a single case of the expected or normal behaviour of the system. With the varying nature of data, it might require an update on the static threshold values of parameters being used in their respective approaches.

In [52], Leandro et al., perform a 3 standard deviation test on the percentile time-series window. This test being a statistical method might not be feasible for a large scale system such as a microservices environment when the data is voluminous with multiple expected cases of normal scenarios and window length cannot be fixed when the data is dynamic.

These studies have shown that traditional statistical methods, machine learning, and deep learning approaches can be successfully used for anomaly detection. But most of these techniques have considered different methods for static thresholding on distances or errors or probability, etc., for scoring an anomaly as the final step irrespective of any algorithm used. This static thresholding might not be efficient enough for a large scale dynamic distributed environment when the nature of data is dependent on time.

However, in this thesis, we complement the existing work and propose an automated prediction-based anomaly detection and localization system which is based on an ensemble of time-series based deep learning model and unsupervised learning algorithms a dynamic thresholding method that is capable of (i) detecting anomalies without any requirement of the explicit set-up of thresholds or static anomaly scoring, and (ii) locating the root cause of the detected anomalies. Also, we are using tracing events data from a large-scale distributed system such as microservices unlike most of the existing literature where the related work on time-series anomaly detection using the distributed tracing data in the microservices system is limited.

Chapter 4

4 Problem Analysis of Cloud System

In this chapter, we discuss in detail about the cloud system of our industry collaborator to better understand the problems faced by them or in general by other cloud providers. This chapter also describes the problem faced by our industry partner dealing with monitoring the performance of the cloud hosting several enterprise applications.

4.1 Cloud System

This research work was developed for a third-party cloud platform for monitoring the performance of its microservices. As discussed in Chapter 2, microservice is composed of loosely coupled services that communicate with each other through service endpoints. In the third-party cloud system, there are more than 95 major microservices that call the backend API. Figure 4.1 depicts the routing of data requests through microservices or components such as App Services, Dashboard, Datalayer and so on, for any enterprise application hosted on the cloud serving different purposes such as Catalog, Billing, etc.

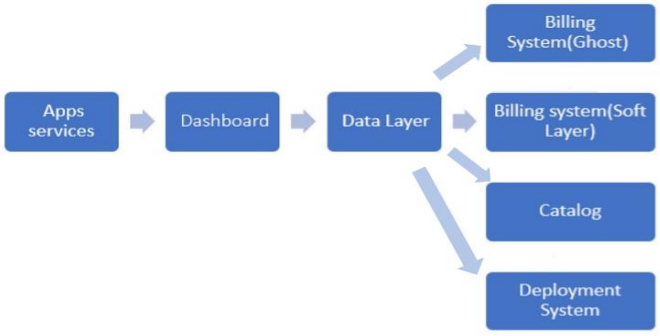


Figure 4.1: Flow Diagram depicting the routing of requests through different services

The clients access the application, the Apps Services requests to open up a Dashboard which renders data from Datalayer by sending a request to it. The Datalayer is a central arbiter that combines data from multiple backend servers and responds to the Dashboard with the response results as shown in Figure 4.2.

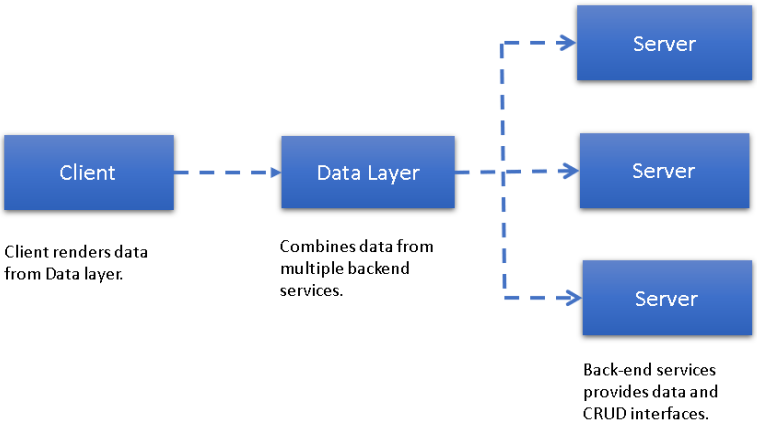


Figure 4.2: Data gateway diagram

The system gets complicated with multiple instances of each of the several services deployed globally. As known, several enterprise applications are hosted on the cloud, increasing the load on Datalayer to access data from the backend services, as shown in Figure 4.3. As mentioned earlier, the loading time of the dashboard of applications accessed by the clients depends on the response time of the data layer. The response time of the data layer further depends on the time taken by all the succeeding components through which the requests were routed for the desired action.

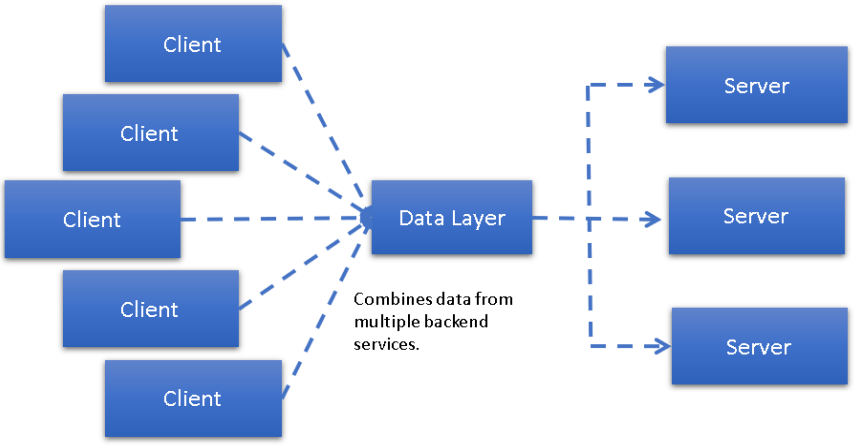


Figure 4.3: For multiple cloud-hosted applications

Out of the many interconnected microservices or components that exist in the cloud system, we select an end-to-end flow of a particular set of services as depicted in Figure 4.4.

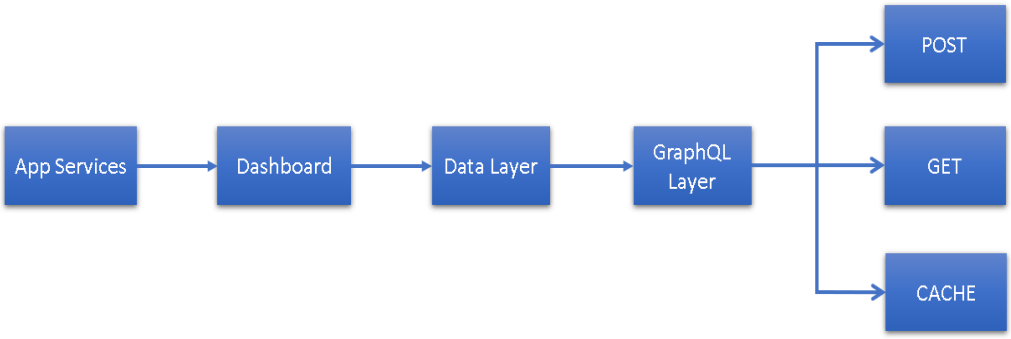


Figure 4.4: Routing of Requests from App Services till GraphQL Layer

To access the Dashboard, a request is initiated which flows through the components as shown in Figure 4.4. The loading time of the Dashboard depends on the time taken by the other succeeding components especially, the GraphQL Layer, which is discussed in section 4.1.1. When the requests flow through these services, the distributed tracing events are generated for every component during a transaction and stored in such a way that the events from the components are grouped under a single ID for a given transaction or request, which is discussed in section 4.1.2.

4.1.1 GraphQL

GraphQL is a query language for an API that provides singular endpoints to the consumer and controls the data flow. With GraphQL it is possible to request specific data instead of asking for all the data from the data sources unlike REST API [89]. Also, to fetch data from multiple sources, only 1 request is required, unlike REST API, as shown in Figure 4.5, and there is no necessity to check which endpoint is needed to get the data as shown in Figure 4.6. GraphQL server accepts both POST and GET requests. POST requests are sent in the form of a JSON object to the GraphQL server. Each request contains a query, or an operation name, or both and may contain variables. A GET request must pass query, operation name, and optional variables in the URL. GraphQL also builds a cache of frequently requested data in order to save the processor time and effort.

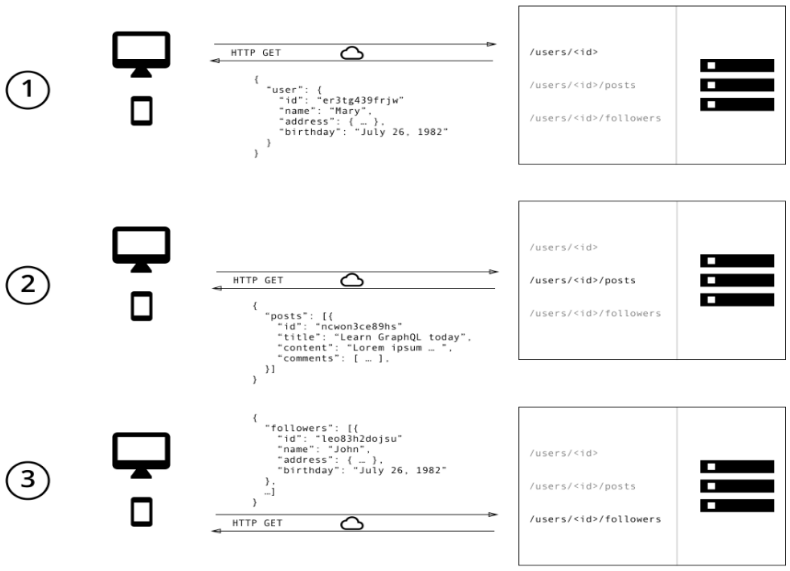


Figure 4.5: With REST, 3 requests are made to fetch data from 3 different endpoints. Also, it over fetches data with additional information. Image Adapted from [90].

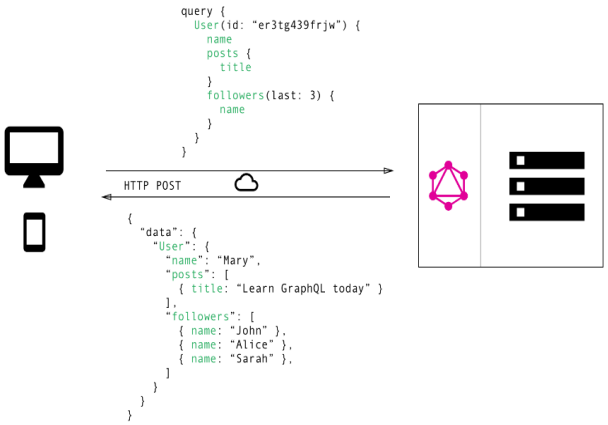


Figure 4.6: With GraphQL, only 1 request is sent to the GraphQL server to fetch the required data. Image Adapted from [90].

4.1.2 Distributed Tracing

Distributed tracing has been considered as a baseline necessity for both software development and operations by organizations [65]. Distributed Tracing, also known as Distributed Request Tracing is a method of understanding the flow of data as it propagates through the components of applications. It profiles and monitors applications, especially

those in a microservices environment, which can be used to facilitate the DevOps teams to pinpoint where failures occur and its root-cause [64]. The difference between Logs and Traces is, logs record important checkpoints when servicing a request, whereas a trace connects all these checkpoints to form a complete path of how a particular request was processed from a client to a server [62].

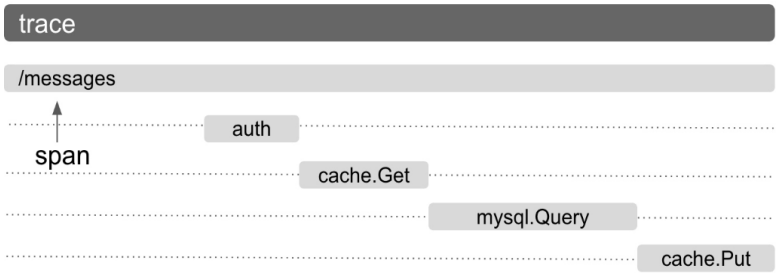


Figure 4.7: Data flow representation of a trace. Image adapted from [67].

Trace - A single trace is a tree of spans that shows the execution path of a single transaction flowing through components in a distributed system [66, 67]. A trace is identified by a unique 16-byte sequence called Trace ID. This Trace ID groups and distinguishes spans. As shown in Figure 4.7, '/messages' is a trace.

Span - Span is the basic building block of a trace, representing an individual unit of work or a single operation in a trace. For example, as shown in Figure 4.7, sub-calls are made to different components like authentication, cache, and database to process the request of fetching a message. Span is identified by a Span ID and belongs to a single trace. Spans contain references to other spans forming a parent and child relationship. A Span without a parent is a root span [67]. A span consists of 11 fields as listed below:

- *Name* - A meaningful span name describing what it does.
- *SpanID* - Span's unique 8-byte identifier.
- *TraceID* - ID of the trace to which a span belongs to.
- *ParentSpanID* - Span ID of its parent or NULL, in case of root span.
- *StartTime/EndTime* - timestamp recording when span operation started and ended.
- *Status* - Integer type code defining logical error model.
- *Time events* - describes that an event happened at a given time during the span's lifetime.

- *Link* - describes cross-relationship between spans in the same or different trace.
- *SpanKind* - details the relationships between spans apart from the parent/child relationship.
- *TraceOptions* - describes if the span is sample or not.
- *Tracestate* - a key-value pair to annotate order/position of request.

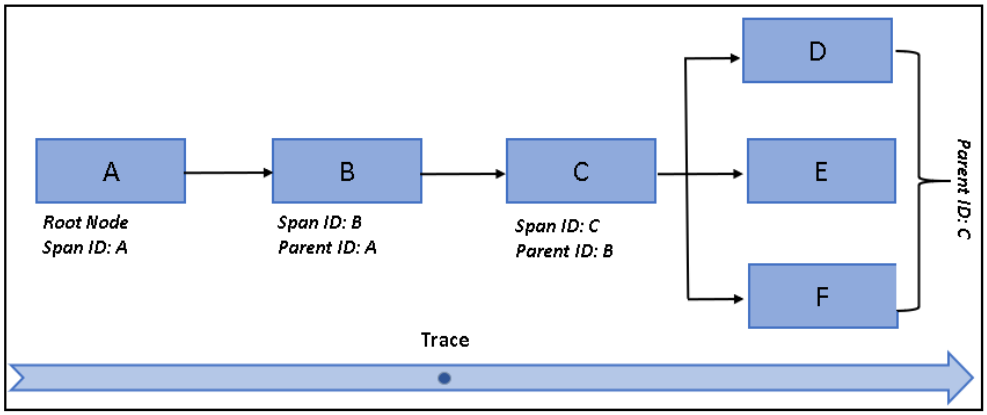


Figure 4.8: Directed Acyclic graph representation of a trace. Each component is labeled with a Span ID and its corresponding Parent ID.

A Trace can be represented as a Directed Acyclic Graph (DAG), as shown in Figure 4.8, where nodes are components denoted by Span ID and edges are the references. Based on this, as per Figure 4.4, the Data layer will be the parent for the GraphQL component, which in turn will be a parent for POST, GET, and CACHE components.

This information can be captured either manually by logging before and after every operation in code or by automated instrumentation. Instrumentation is a process in which applications' code is extended to capture trace spans in the path of processing a transaction or user request [69]. In Automatic instrumentation, a run-time automated process identifies the frameworks and libraries which are in use within an application and instrument those libraries to capture tracing information automatically without requiring any code change [69]. For example, whenever a request is made to the database, a listener will extract and store this information, in an automated instrumentation scenario. Capturing trace manually through logging is not advisable since it is not structured well. There are microservices

Standards and Tools like OpenCensus, OpenTracing, etc., that are used to extract the traces [68].

4.2 Concerns with Existing Monitoring

Existing monitoring systems fire alerts when the key performance indicators exceed a given threshold value. However, this is insufficient when network architecture and/or load changes over time. The Microservices system has different points of failure than traditional codebases and presents significant issues when site reliability engineers attempt to monitor and diagnose runtime issues. Ideally, a site reliability engineer should be able to: (i) visualize the performance of the system, (ii) be notified when the behaviour of the system changes, relative to historical norms, and (iii) be able to quickly identify the root cause when a key performance indicator changes. Each of these requirements is arguably difficult. Existing monitoring also requires the staff to visualize the underlying metrics data (e.g., CPU utilization, Mean Response Time, etc.) using different visualization tools such as Kibana, for analyzing and alerting agents upon violation of conditions or threshold. “The staff are concerned that when an issue is encountered, the peaks on the graph shoot off and often require the operations team to manually validate whether it is an anomaly or not. The staff gets multiple alerts at odd times and most of the time the parts are not critical and turn out to be false alarms. The staff mention that they do not do very much with the data generated by the microservices and are keen on reducing the burden of the monitoring process that is being carried out.”

4.3 Problem Analysis of Monitoring Microservices System

As mentioned in Chapter 1, there is a tremendous increase in the migration of monoliths to microservices due to the several advantages of the later discussed previously. However, it has increased the complexity of the existing monitoring framework. Apart from the microservices such as Dashboard, Catalog, etc., shown in Figure 4.1, there are more than 95 major microservices that call the backend API as mentioned earlier. Figure 4.9 depicts how complex a large scale distributed system such as a microservices environment gets with multiple applications triggering requests from one microservice to other microservices to access the backend. For example, the performance (response time) of one microservice

depends on the load and its succeeding microservice, i.e., the time taken by microservice ‘n’ to respond to a request depends on the processing time of its succeeding microservice ‘n+1’, which in turn depends on the processing time of microservice ‘n+2’ and it goes so on. When there is an increasing number of applications being accessed by the clients, the response time of the microservices increases, i.e., due to the high load or high volume of requests, the time taken by each microservice to process the requests increases, causing cascading effects in the response time of the preceding services to respond the client with the required information.

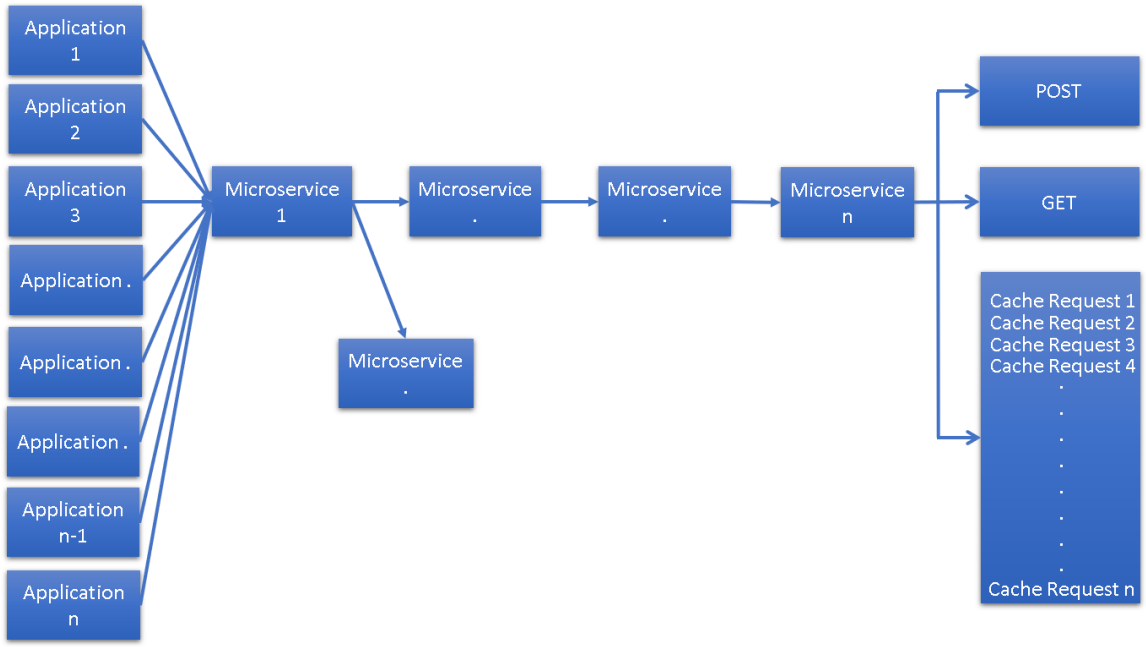


Figure 4.9: Multiple Clients or Applications triggers requests to multiple Microservices

Similar to the above example, several such performance problems occur in a large-scale distributed system, differing in the way they manifest themselves and symptoms that they show. There are various performance issues due to High CPU load, CPU Thrashing, Memory exhausted, I/O bottleneck, Slow Disk I/O, Too many disk I/O operations, Deadlocks, Algorithmic complexity, etc. These performance issues are identified by different metrics (or) KPIs such as response time of services, error rate, throughput, workload, resource utilization, etc. With various issues being generated several performance anomalies (such as high response time, high error rate (or) high network

utilization) go undetected which might result in potential failure or outages, impacting the stability of the cloud system and user experience.

In microservices, each service and its multiple instances needs to be monitored because one service creates multiple copies (instances) to make sure that if one goes down there is always another copy of that service available. Therefore, knowing the state of a single instance of a single service isn't enough for monitoring a given and hence, metrics of all the instances of a given service need to be aggregated. Secondly, in this system, there is a need to monitor multiple services at once, and these services might even use different technologies. There is a need to check the logs of multiple services and track one user request through multiple systems.

As per the existing monitoring framework described in Section 4.2, each service is monitored based on a thresholding approach, where an alert is fired when the key performance indicators exceed a given threshold that is set up using a 3-Standard deviation rule. With a rule-based thresholding approach, the existing monitoring framework might not be reliable to monitors all the instances of all the services at once as discussed above, due to the large volume of data and also might be inefficient when the data or load varies over time. For example, the average response time of a given service today might be 1000 ms, but when the request load changes or when a new update is performed on the system, it might speed up the processing resulting in average response time to be less than 1000 ms. Hence, given the dynamic nature of data, this static threshold might require an update. But, for a cloud system with more than 95 services, it becomes tedious and time-consuming to do so.

Apart from detecting an issue (an anomaly), analyzing the given issue and identifying the causal components for the same is arduous in a microservices environment because of the complexity of communications between different microservices. A single user action triggers a chain of downstream calls to different microservices as they pass data back and forth from a client to its server. In this third-party cloud system with more than 95 microservices and its multiple instances, it becomes challenging for the SREs analyze and to track the root-cause or faulty microservice/component.

Hence, to tackle the aforementioned problems, we require an automated anomaly detection and localization system that can detect the performance anomalies without using static threshold rules. Thus, this would reduce the manual burden of monitoring the services visually, and locate the causal components to determine the root cause of the anomaly. Also, this would ease the process of monitoring the microservices-based cloud system to maintain the health of the cloud system.

Chapter 5

5 Research Methodology

In this section, we describe the strategy of our solution and the system development methodology conducted in our research work.

5.1 Solution Strategy

Our research emphasizes on easing the monitoring process of a microservices-based cloud system by developing an automated prediction-based anomaly detection and localization system, (i) which detects performance anomalies using a time-series deep learning model and an ensemble of unsupervised learning techniques that can handle a huge volume of data generated from each of the individual microservices and avoid the burden of static thresholding approach that is used in the existing monitoring framework and other literature works as discussed in Chapter 3 and 4, and (ii) identifies the casual components of the detected anomalies.

As mentioned in Section 4.3, there are more than 95 major microservices such as Dashboard, Catalog, Datalayer, etc. Out of the 95+ microservices' data, we consider a few microservices such as App Services, Dashboard, Datalayer, and GraphQL, as shown in Figure 4.4, for our analysis in this work and try to detect the performance anomalies of the GraphQL service by using the proposed detection and localization system.

Data Decision: In a microservices system, it is important to see how individual services communicate and how requests flow through a specific combination of services [62, 63] to understand the path of a given transaction. Hence, we use the distributed tracing events generated by each service for our analysis which records the network operations for a given request to understand the flow of the transaction. As discussed in Section 4.3, the tracing events of all the instances of a given service(s) are aggregated and stored at a centralized location, in the Elasticsearch system of the production environment, which is further extracted using five different python scripts, discussed in Chapter 6. This work uses the real-world production environment data of the third-party cloud system.

Meta-Data: The tracing events in the form of JSON objects have several data fields out of which we describe a few, as tabulated in Table 5.1, that we use in our work. The metadata typically contains the following.

Trace Event JSON Data Fields		Description
traceID		Transaction ID - A 16 byte ID used to uniquely identify a set of network operations
Name		The URL associated with the network operation
Id		A 16 byte ID used to identify the current / a single network operation
Kind		Differentiates between incoming (SERVER) and outgoing (CLIENT) calls
Timestamp		The time the operation commenced
Duration		The time required to complete the operation
Tag	http.host	Network identifier for target service
	http.method	POST/ GET/ NULL
	http.status_code	A predefined integer indicating the success or failure of the operation
	environment	type of enviroment (production/development/testing)
	localEndpoint.servicename	Microservice name
parentID		The span ID of the network operation that caused the current span

Date	Date of when the transaction happened

Table 5.1: Distributed tracing fields of microservices

Data Analysis: The trace events were analyzed to select all the possible data fields that we could use for monitoring the services. Post analysis, we found that the duration, i.e., the time taken by service to process the incoming request is the main key performance indicator for our work. While tracking the flow of a given transaction or trace ID through different spans (see Figure 4.4), we realize that we do not want to find the anomalous individual transaction that is taking more time, but we want to determine the instance when a particular service (say GraphQL service) takes more time to process its incoming requests. Therefore, we selectively take the processing time of the GraphQL service for a single request.

Multiple transactions are processed simultaneously at a given time, hence we group the transactions in a 5-minute interval. Therefore, for every 5-minute interval, all the Trace IDs are grouped to calculate the total number of requests per interval and sum their processing time by GraphQL service. The performance metric used for analyzing the performance of GraphQL service is the average response time, computed using the above two features. The reason for choosing a 5-minute interval is stated below.

5-minute interval	**5-minute interval**	**5-minute interval**
Requests: 10 Total Time: 5 s	Requests: 10 Total Time: 13 s	Requests: 10 Total Time: 11 s
Avg. Res. Time: 0.5 s	Avg. Res. Time: 1.3 s	Avg. Res. Time: 1.1 s
Avg. Res. Time: 0.9 s		
Avg. Res. Time: 0.96 s		

Table 5.2: Reasoning for 5-minute interval decision

For example, let's assume the average response time of a service is 1 second. As shown in Table 5.2, when a 5-minute interval is considered, the average response time of the last 2 intervals gets detected as anomalies (red-highlighted) since it exceeds more than 1 second. However, if a 10-minute or a 15-minute interval is considered, the average response time

doesn't get detected as anomalous, since the total time gets divided by the load or large of number requests. And if a 1-minute interval is considered, the system would detect too many false positives. Hence, we decide to choose a 5-minute interval.

During the initial stages of our work, we conducted exploratory analysis on a daily basis through scatter plots, histograms, bar graphs, etc., to visualize the pattern in the dataset.

Anomaly Detection: We performed detection for the two computed features (Total number of requests, Total response time for every 5-minute interval) using unsupervised learning algorithms such as K-means Clustering, which groups data into clusters and identifies the points that are away from the clusters. We also experimented with Gaussian Markov models, KNN, Isolation forest, OC-SVM, Angle-Based, and Cluster-Based Outlier Detection to identify the outliers using the two input features. As we collected data for over a week, we observed that the 'total number of requests' feature followed a sequential pattern that varied with time. The aforementioned unsupervised algorithms fail to learn the temporal characteristics of the sequential data and hence, time-series models were tested for our dataset.

We engineered a new feature "Average response time" of GraphQL using the 2 attributes – "Total number of requests" and "Total duration", to learn its timely pattern and create a baseline for the average response time of the GraphQL service. When trace events were collected for over three weeks or a month duration, sufficient enough to capture the 'daily' and 'weekly' patterns in the data, we experimented with statistical time-series models such as ARIMA (Autoregressive Integrated Moving Average) which works for a stationary time-series data, i.e., mean and variance are constant over time.

An Augmented-Dickey Fuller (ADF) test was conducted to check if the time-series data (input data: Average response time, timestamp) is stationary or not. Since the data is stationary with seasonal and trend patterns, it was converted to non-stationary for testing the ARIMA model. SARIMA (seasonal ARIMA) was tested as well. Both the ARIMA and SARIMA models yielded poor results and took a long time to process the data. Such models allow only one independent variable or one feature such as Average response time alone.

When the time-series data were decomposed into STR components (Seasonality, Trend, Residue), after the seasonality was removed, we observe the ‘trend’ to follow an irregular pattern/non-linear pattern. To build a statistical method that can describe this time-series data, one would need to build a piece-wise function. This might be expensive to learn when there is a large volume of data.

Considering the advantages of LSTM over ARIMA, SARIMA with respect to processing speed, no pre-requisites of non-stationary data, allows multivariate data, and the ability to handle long term and non-linear time-series data, we choose LSTM for time-series forecasting. The predictions made by the algorithms are further utilized for the detection process, which is discussed in Chapter 6.

Localization: For the localization process, we collect the Id, Parent ID, duration, http.method, service name, http.host data fields from the trace events for all the microservices to understand the communications calls across individual services. We use ‘Networkx’ python library to analyze and understand the network or the directed graph structure connecting different microservices. The intrinsic details of how the casual components for an anomaly are located are described in Chapter 6.

5.2 System Development Methodology

We adopted agile methodology to conduct our academic research where we broke the entire process of research into various stages (described below) that were incremental and iterative, rather than following a waterfall approach since it is not flexible for researching as it gets complicated while following a sequential series of events and fails to adapt to any new requirements with early delivery of small incremental builds. For example, after conducting a literature review, analysis when we build a prototype and start the implementation, we cannot improvise the prototype model or perform a literature review again while we perform the implementation simultaneously. In research, these stages need to be iterative and incremental.

Our agile research protocol involved the following stages:

1. **Split the research activities –** The activities were split into academic research and a practical approach to solving the problem posed by our collaborator.
    a) **Requirements Gathering from Industry partner -** In the preliminary stage, we acquired the information from our industry collaborators regarding their problems or requirements and goal (see Section 1.2) We gathered information about the cloud system, system functioning, and the performance monitoring issues that they were facing.
    b) **Performing Systematic Literature Review (SLR) -** Based on the information provided we started performing a systematic literature review. A broad search was carried out to look for what and how different monitoring problems were approached in the existing research works across various domains and not limited to the computing field. We shortlisted research papers that were dealing with Machine Learning since the old technologies or traditional approach to solving those problems were outdated or not practical in the current times for dealing with time-series data in a microservices environment [38, 39]. We later filtered the papers by introducing the combination of specific or similar keywords such as 'Anomaly detection', 'Microservices', 'Cloud computing', 'Monitoring', 'Application performance management', 'Machine learning', etc. The study on existing works was organized into a spreadsheet to keep track of papers that were reviewed and the gathered related work was analyzed for research gaps.
    c) **Building Prototype -** During the initial stage, based on the literature review and gaps analyzed, we shortlisted a few approaches of ML to develop an initial prototype or high-level design of the approach as discussed in the previous Section 5.1. To test the applicability of the idea we tested the approach on a sample data set before we gathered the actual data. Based on the results we achieved using several approaches, we improvised the design of our initial proposal iteratively and incrementally.
    d) Setting up Infrastructure - Simultaneously, the infrastructure was set up by our collaborator where a cluster in Elasticsearch System was configured to

collect data or tracing events from the microservices through Redis while they were running in the production environment.

e) **Data Collection and Analysis -** During the preliminary stage, the data collection process was carried out every day since it was a production-based data and we did not have enough historical data to carry out the anomaly detection. While the data was incrementally gathered, we processed the data which was in the form of JSON objects, and analyzed the patterns via visualization tools and graphs to better understand the nature of data to choose the right approach. Over 900GB of pickle object files of trace events were collected over 5-6 months from June to December 2019. Pickle module helps to serialize python object structures like list, dictionaries, etc., into a character stream before writing to a file. Using the character stream the python object can be reconstructed during de-serialization of the objects using the pickle module.

f) **Implementation & Evaluation –** During the initial stage, we implemented basic unsupervised clustering algorithms, KNN, etc., to detect anomalies in the data, and during the intermediate stage when we had collected enough data we developed time series models such as ARIMA, SARIMA, Autoencoders, since our data is sequential with temporal attributes as discussed in the previous Section 5.1. Later with the help of domain experts, we manually labeled the data to evaluate our approach. Based on the assessment of results, feedback during sprint review meetings, and simultaneous study on existing work we improvised our prototype at several stages until we finalized the current proposed detection and localization system.

2. **Sprint Planning** - We conducted bi-weekly meetings with our supervisor and industrial partner for a duration of about 30 minutes to an hour during initial stages to brainstorm the ideas from both academic and industry point of view and discuss the small goals for the next activity(Stage 1 activities) and plan out its duration.
3. **Sprint Review Meeting** - During the review meeting we discussed the results of our work carried out and about the technical challenges or difficulties encountered during the work, and brainstormed ideas on what could be done next.

4. **Weekly Meetings** - During the preliminary stage, we conducted meetings for giving KT (Knowledge Transfer) or training us about the system. We had short meetings with our collaborator to update on the ongoing tasks and discussed any issues that we were facing.

All the above stages were performed in an iterative and incremental manner.

Chapter 6

6 Proposed Solution: Anomaly Detection and Localization System

In this chapter, we describe our proposed automated system to utilize the distributed tracing events of microservices on the cloud to detect anomalies in the performance metric data through monitoring the Key Performance Indicators present in the tracing events and further locate its causal component which will facilitate the DevOps or IT operations team to take appropriate actions to fix them and maintain a healthy and stable cloud environment.

6.1 Generic System Design of the Proposed System

This research proposes an automated anomaly detection and localization framework, which collects the distributed tracing events, which is discussed in the later section, generated by the running microservices on the cloud, analyzes them to detect the performance anomalies in it and identifies the causal or faulty components for the detected anomalies. Figure 6.1 provides a generic prototype of the proposed detection system, depicting how the control is going to flow from one component to another to detect the issues in the microservices-based cloud environment.

The data generated by microservices such as logs, metrics, tracing events that fall under the categories of both structured and unstructured data are collected and used for further analysis. Unsupervised machine learning techniques are applied to the collected data which will identify and learn the patterns from different metrics of data such as Network usage, Request Arrival, CPU usage, etc., and build a prediction model to generate the underlying baseline for each of the performance metrics. This predictive model is further used to make future predictions on the unseen data or the new incoming stream of data in real-time. As the incoming stream is fed to the model, the model detects the data points that are deviating from its learned behaviour or normal behaviour, labels them as anomalies, and sends the results to the Alert Management system to take further actions.

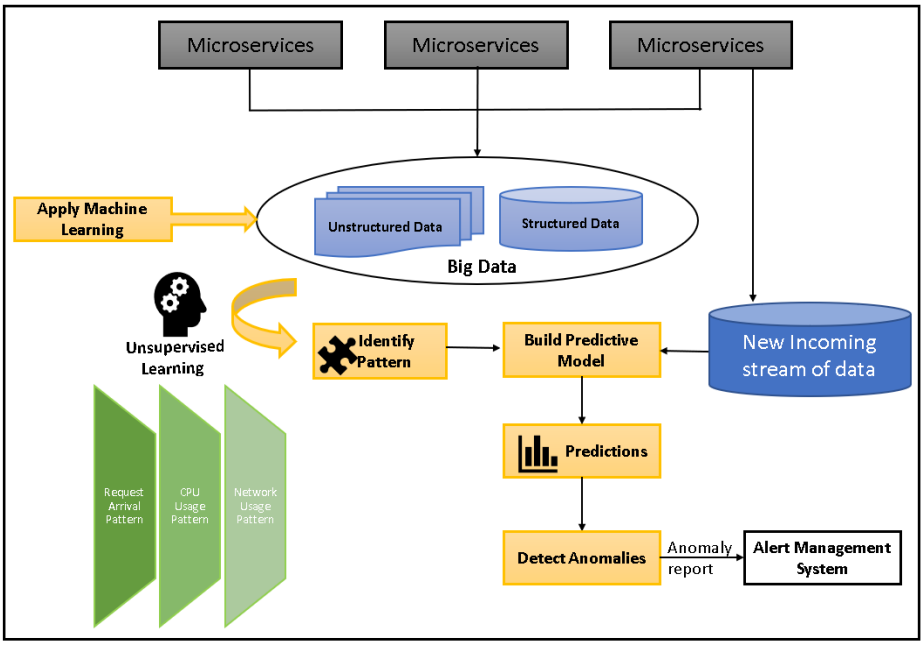


Figure 6.1: Generalized layout of the proposed detection system

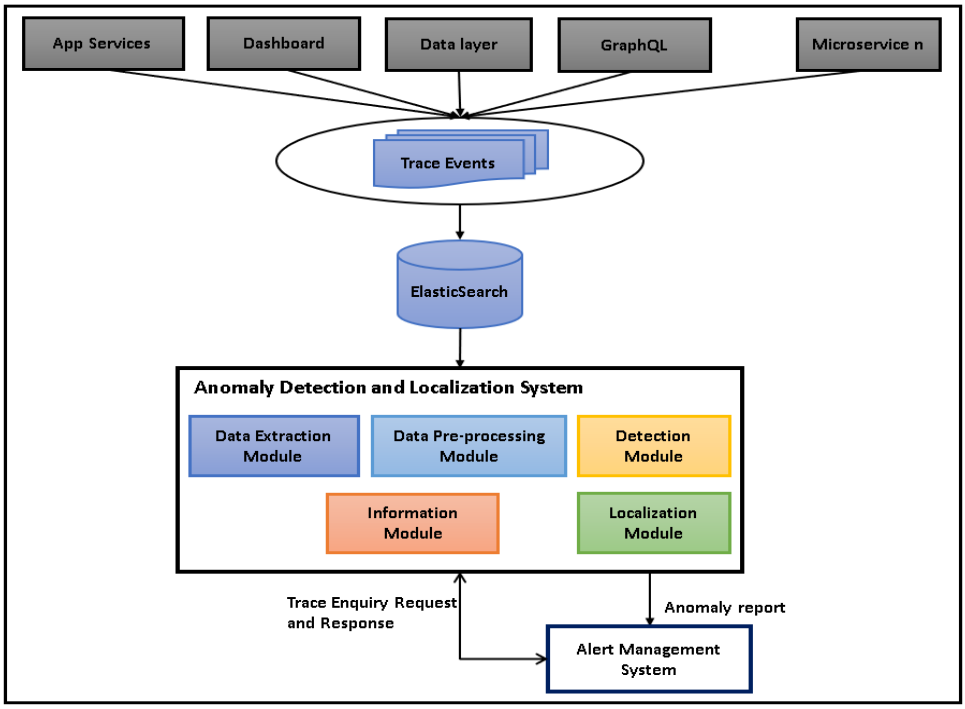


Figure 6.2: Context Diagram of the Proposed System

The microservices used in our work are the components - App Services, Dashboard, Data Layer, GraphQL Layer, and sub-calls to POST, GET, Cache from GraphQL, that were discussed in Chapter 4. Figure 6.2 shows the context diagram of the system where the proposed black-box anomaly detection and localization system fits. When the requests flow through these microservices (refer Figure **4.4** as well), the tracing events are generated for every service or component during a transaction and stored in the ElasticSearch system which is used for further analysis by the proposed black-box system as shown in Figure 6.2. The proposed system further sends an anomaly report to the Alert management system when it identifies and locates the anomalies and its causal components respectively.

6.2 Detailed Layout of Proposed System

The intrinsic details of the proposed system are shown in Figure 6.3. The proposed system has 5 modules - Data Extraction Module, Data Pre-Processing Module, Detection Module, Localization Module, and Information Module. The data flow starts from the Data Extraction module, where the tracing events of different services are extracted from the ElasticSearch (ES) system and further processed using the Data Pre-Processing module. The data is further fed to the Detection module and Localization module to detect and locate the root cause of the anomalies respectively. The functionality of each module is explained in detail as follows.

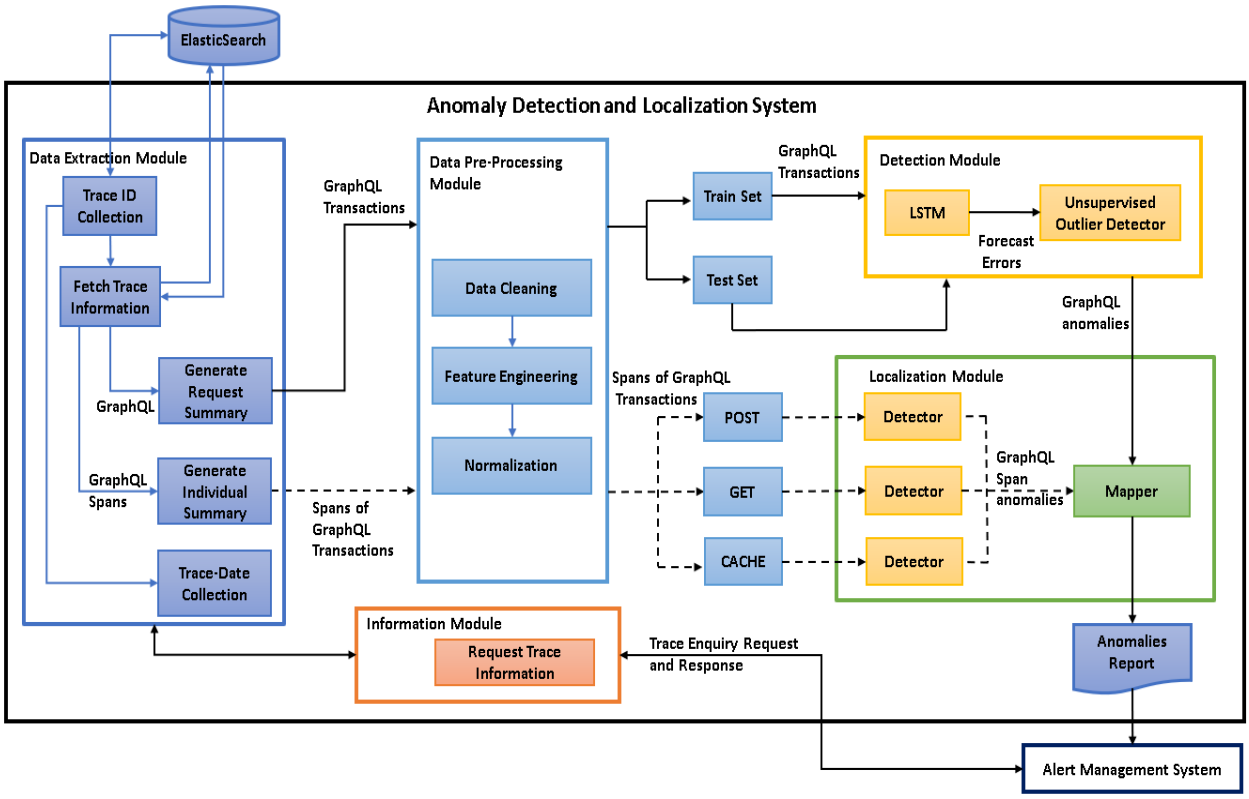


Figure 6.3: Detailed layout of the proposed anomaly detection and localization system

When the request flows through Apps Services -> Dashboard, Dashboard -> Data Layer, Data Layer -> GraphQL Layer (see Figure **4.4**), the tracing event data is captured at each span/component during this transaction. There could be several such transactions happening frequently and flowing through a different set of microservices before reaching the GraphQL Layer. We capture all these transactions where requests are made to the GraphQL layer on a daily basis to collect the data. Each transaction is stored in an index named “cloud-datalayer” in the Elasticsearch (ES) system as shown in Figure 6.2. The data captured includes a stream of tracing events generated by OpenTrace library which includes trace fields such as Trace ID, Span ID (each step in a transaction), Parent ID, duration, status code/error message, annotation, etc., which was discussed in Section 4.1.2. The metadata or the data fields of the tracing events used in this work are described in Table **5.1**.

6.2.1 Raw Data

The raw data stored (refer Table **5.1**) in the ElasticSearch system (ES) is extracted into the local system for further data pre-processing and analysis. In this section, we discuss the raw data or the tracing events stored in ElasticSearch system that is being used for our work. To select transactions that have made calls to GraphQL, a filter for 'name' field equivalent to '/datalayer/graphql' is applied under the index "cloud-datalayer" from ES cluster. Figure 6.4 shows the event information with distributed tracing fields for the GraphQL layer for a given range of timestamps. These events have data fields such as Trace ID, ID/Span ID, Parent ID, Duration, Error codes, etc., which were discussed in detail in Section 4.1.2. Every record in the figure is an individual span belonging to their associated distinct transaction/trace ID. For example, the first record in Figure 6.4 is a span for the red highlighted trace ID or transaction.

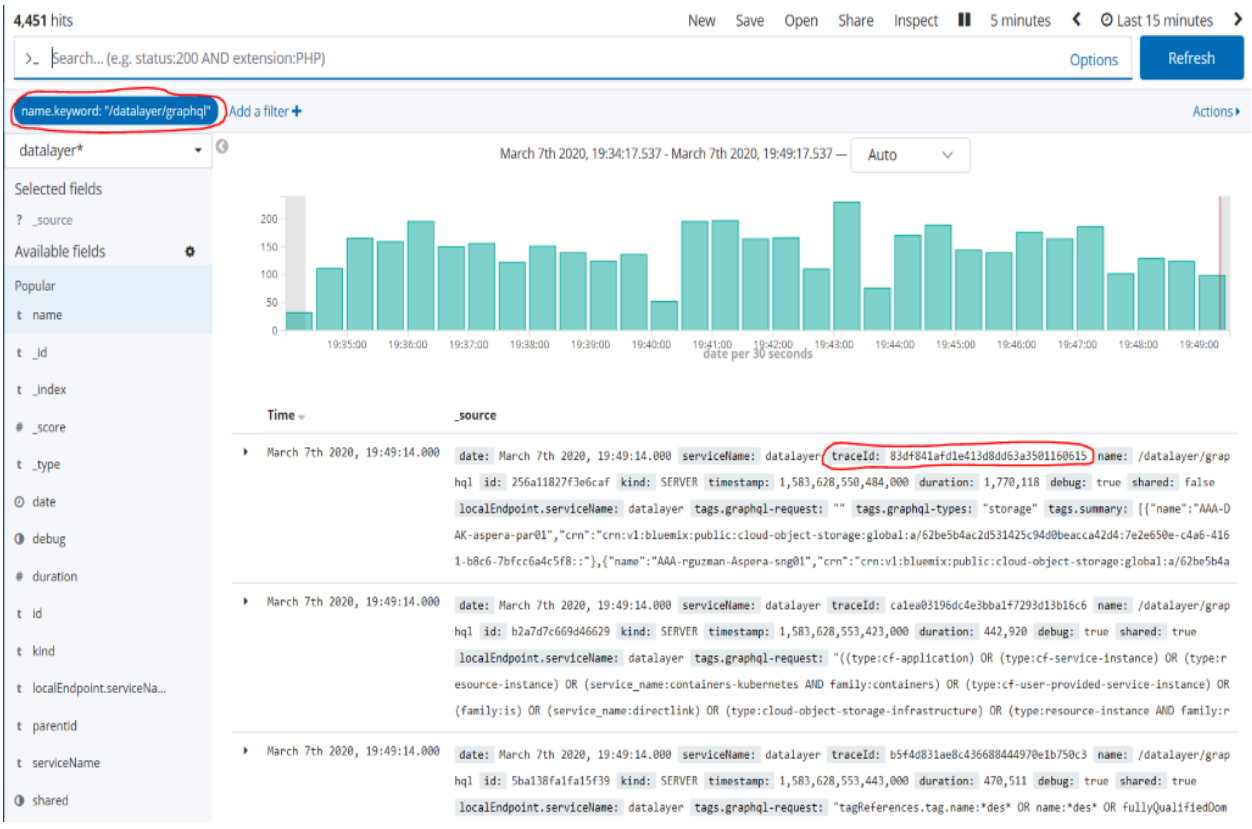


Figure 6.4: Trace data stored in the ElasticSearch cluster displayed using the Kibana tool. Under index: "Cloud-Datalayer" or "datalayer*", a filter is applied for name: "/datalayer/graphql" to check the distinct transactions or requests to GraphQL Layer.

On applying the filter for the ‘traceId’ field with this highlighted trace ID, we can check the detailed description of the flow of calls through various components for the highlighted transaction ID as shown in Figure 6.5. We can selectively choose the fields that we are interested in looking at and as depicted in the figure we can trace out the flow of an entire transaction by using parent ID associated with every span or ID. All the rows in Figure 6.5 are the spans or sub-calls to complete one transaction.

Every transaction has a varying number of spans that takes a certain amount of time to complete its execution. In one minute, hundreds of thousands of transactions/requests get processed. In this work, the transactions or trace IDs are grouped for every 5-minutes interval.

1 minute -> ‘n’ requests (or) transactions
1 transaction -> ‘m’ spans

Figure 6.5: Displaying the tracing information for one trace ID or transaction.

6.2.2 Data Extraction Module

The Data extraction module is responsible for extracting the raw data or trace events from the ElasticSearch (ES) system for all the services or components. In this section, we describe how data extraction is carried out using 5 steps.

1. **Trace IDs Collection:** The trace IDs of the transactions routing through the GraphQL layer is collected from “cloud-datalayer” index for every 5-minutes interval. The Trace IDs collected are stored in separate pickle object files for each 5-minute bucket in our local system instead of JSON files in order to save the storage space and for faster processing. Since the data is collected for every 5 minutes interval, there will be 288 files generated per day.
2. **Fetch Trace Information:** The pickle object files generated from step 1 is loaded to read the Trace IDs and fetch all its detailed information, i.e., all its span information from the “cloud-datalayer” index using the ‘traceId’ field as a filter. The ES system returns a maximum of 10,000 JSON objects for any given input query. But if there are say, 2000 Trace IDs in any given 5-minute interval and if each Trace ID has at least 20 spans, then there will be 2000*20 = 40,000 JSON objects/ span details to be fetched from ES for given single query. Hence, to handle the size restriction, a simple concept of slicing is used, which slices the TraceIDs list from every input file into different smaller bucket lists which are further passed to the function to triggering the ES system to fetch trace information one by one. An email alert is sent once all the trace ID’s detailed information using 5-minute interval files is fetched as given in Figure 6.6. This process also generates 288 files per day.

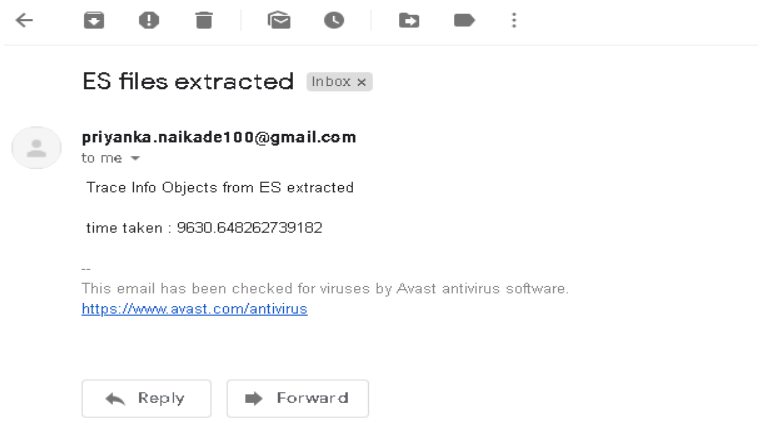


Figure 6.6: Email Alert after fetching trace information data from ES.

3. **Generate Request Summary Information:** All the files generated from Step 2 above is used to generate summary data of GraphQL requests. Summary data for GraphQL indicates the Total number of Requests, Total Duration, and Status codes for every 5-minute interval for GraphQL component alone. One trace indicates a single request. Hence, the total number of traces is counted, and the duration of each trace is aggregated for every 5-minute interval. The summary data is then stored in a single pickle file object which is later used in the anomaly detection module.
4. **Generate Individual Summary Information:** All the files generated from Step 2 is also used to generate summary information similar to Step 3 for each sub-calls or sub-components of GraphQL. By sub-components, we mean the GET, POST, and Cache requests from the GraphQL layer as seen in the previous chapter. These 3 individual requests from GraphQL are grouped into 3 buckets: GET, POST and Cache, and assumed to be individual components as an analogy to successive components. This summary information for 3 sub-components is stored in a separate pickle file object for each sub-component which is later used in the localization module.
5. **Trace-Date Collection:** The individual files generated in step 1 that has a list of Trace IDs for specific 5-minute intervals or date timestamp is utilized to aggregate the trace IDs across all the timestamps. This file is used in the Information module as a part of the validation to check if a particular TraceID exists or not, and provide the details of the occurrence of a trace if it exists. It consumes a significant amount of time to iterate through all the files generated for validation or to check when a particular transaction happened. Hence, a list of dictionaries is created which stores dates/timestamp of the 5-minute interval as keys and Trace IDs as values for the keys for a quick lookup of trace IDs.

Over 900GB of pickle files were collected on a daily basis over 5-6 months so that there is enough data to train the deep learning model.

6.2.3 Data Pre-Processing Module

The unstructured summary data generated and stored in pickle file objects in the Data Extraction module is read, formatted, and loaded into data frames in Python notebook for further pre-processing and analysis. The ‘Response/status codes’ feature clubbed together as a single JSON object is split into series, making each status code a different attribute in the dataset. The date interval which doesn’t have any values (with NaN) for a given status code is filled with 0’s.

Date	Requests	TotalDuration	ResponseCodes
2019-12-29 12:30:00	1223	728365.917	{'200': 2534, '404': 2}
2019-12-29 12:35:00	1381	720363.817	{'200': 2823, '500': 22, '501': 1, '502': 1}
2019-12-29 12:40:00	1240	621707.189	{'200': 2533, '404': 1, '502': 1}
2019-12-29 12:45:00	1450	793582.986	{'200': 2942, '503': 1, '404': 1, '502': 2}
2019-12-29 12:50:00	1587	786777.113	{'200': 3253}

Figure 6.7: Data loaded in a Dataframe

AvgDuration	200	403	503	404	400	301	500	502	401	504	429	202	501	523	422	413	522	521	405	530	526
757.793027	4870.0	15.0	1.0	64.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
749.124397	3759.0	11.0	0.0	43.0	0.0	1.0	4.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
810.259289	4780.0	11.0	0.0	50.0	2.0	0.0	1.0	2.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
700.671463	4538.0	21.0	0.0	86.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0
752.800528	3520.0	25.0	0.0	20.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Figure 6.8: Average Response Time calculated and Status codes split into series are loaded in the data frame

Feature Engineering: New features or attributes are engineered from the existing features as shown in Figure 6.9. Since the network operation data is grouped into 5-minute buckets, the average response time of GraphQL for the 5-minute interval is used as the performance indicator. The Average Duration or Average Response Time is calculated as Total Duration/Total Requests for every 5-minute interval. New features are engineered out of the timestamp such as the day of the week, daylight or night time, the hour of the day, weekday or weekend, holiday or non-holiday. Additional features such as average response

time 5 minutes back, 1 hour back, 1 day back from the current timestamp are added by shifting the values of average duration by 1, 12, 288 values respectively. These new features are added since this will help a model to better interpret the patterns or represent structures or seasonality in the data.

hours	daylight	timeofDay	DayOfTheWeek	WeekDay	categories	Date_only	is_holiday	5minback	1hrback	1dayback
0	0	1.0	1	1	2	2019-06-18	0	0.000000	0.0	0.0
0	0	1.0	1	1	2	2019-06-18	0	757.793027	0.0	0.0
0	0	1.0	1	1	2	2019-06-18	0	749.124397	0.0	0.0
0	0	1.0	1	1	2	2019-06-18	0	810.259289	0.0	0.0
0	0	1.0	1	1	2	2019-06-18	0	700.671463	0.0	0.0

Figure 6.9: New features engineered

Feature Scaling: All the features in the dataset are on drastically different magnitude scales. For example, the value of the 'total number of requests' is '1223' and the value of 'total response time' is '728365.917' as shown in Figure 6.7. Features with differing scales can impact the machine learning process where one feature can have more influence than the other. Feature rescaling is done by normalization which makes sure that all the features are given equal importance [107]. Normalization makes the optimization process of a neural network smooth [47]. In this dataset, min-max normalization is used where the features are scaled to [0,1] range so that features have a positive range of values rather than standardization which produces [-1,1] values, which is not applicable for features like response times. Min-Max normalization is formulated as given in Equation 6.1 [47], where x' is the normalized value and x is the original value of the feature.

$$x' = \frac{x - min(x)}{max(x) - min(x)} \tag{6.1}$$

The pre-processed data extracted from a large number of trace events is next used in the Anomaly Detector Module.

6.2.4 Anomaly Detector Module

The Anomaly Detector module detects the performance anomalies by using the proposed novel combination of a deep learning model and an unsupervised learning algorithm to generate a baseline for the expected average response time of the GraphQL component and detect anomalies when data deviates from the expected baseline. Firstly, a time-series based prediction model is built using LSTM to learn the pattern of the time-series data and forecast future values of the features. Later, anomaly detection is performed by applying an unsupervised learning approach to prediction errors. The working mechanism of this module is explained in detail below.

6.2.4.1 Training and Testing Datasets

The dataset used for building the model is a real-world production dataset. The detection approach is based on the assumption that the majority of the data is normal and only a few anomalies exist in the historical data collected. The data is split into training and testing sets. The data used for training is expected to be normal i.e., without any anomalies. Since the data was captured from the production environment, the data consists of anomalies. Training the data with anomalies is not appropriate as the model has more chances to learn the anomaly pattern and would fail to detect anomalies when it encounters one. Hence, for training the model we used the portion of data which looks normal based on visualization of 'Average Response Time' feature of the GraphQL service (for example from 21st September to 30th October 2019) as depicted in Figure 6.10, and the last portion of the data is used for testing purpose.

For the time-series prediction model, two variants of data: Univariate data, which consists of a single feature - "Average Response Time" and Multivariate data which consists of multiple features is used for our experiments which are discussed in Chapter 7.

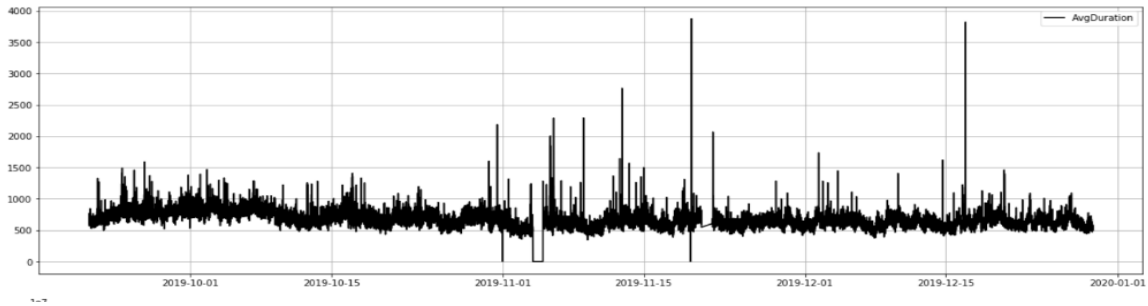


Figure 6.10: Average response time (in ms) of the GraphQL component plotted against the x-axis timestamp

6.2.4.2 Time-Series Forecasting Model

LSTM model is used for the time-series forecasting. For a time-series prediction, the unsupervised problem, i.e., there are features, say Average Response time without any labels, is converted into a supervised problem by partitioning the time-series data into two features, one as input sequence and the other as the target sequence. The partitioning process requires two parameters to be set, namely, *lookback as l* and *future_steps as f* using which the model predicts the next *'f'* values by looking back past or previous *'l'* values. For instance, given an unlabeled sequence of time-series values T1, T2, T3, T4, T5, T6, T7, T8, T9, T10, for a *lookback*=3 and *future_steps*=1, conversion from unsupervised to supervised takes place as follows:

<<Input>> ---> <<Target>>

T1, T2, T3 ---> T4
T2, T3, T4 ---> T5
…….
…….
T7, T8, T9 ---> T10
T8, T9, T10 ---> T11

The model learns using the input sequences and target sequences, where past 3 timestep values are used to predict the next 1 timestep value. For the given input data of 'T1-T10', the model predicts 'T11' as highlighted above. Both the training and testing dataset is partitioned by specifying these two parameters. The model building involves 5 steps – Defining, Compiling, Fitting, Evaluating, and Making Predictions.

To define a network means setting up the model architecture for which an instance of a Sequential model is initiated so that all the layers in a network are stacked sequentially. The layers are then created and added in order. The input sequence of partitioned time series or training data is reshaped into three-dimensional data, which comprises samples, timesteps, and features and fed as an input to the first layer. We then add other hidden LSTM layers depending on the requirement. The network usually consists of multiple hidden LSTM recurrent layers with a different number of units followed by an output layer which is a fully connected dense NN layer used for outputting predictions. The number of units in an output layer is the same as the *future_steps* value, with one neuron for each future value. The dropout layer is used between two consecutive recurrent layers to prevent over-fitting. Over-fitting is a modeling error where the model works well on a training set but performs poorly on a testing set.

Once the network is defined, it is then compiled. The sequence of layers is transformed into a series of matrix transforms during the compilation process. It requires certain parameters to be specified for training the network, such as the optimizer and loss function parameter. The loss function evaluates the network to determine its loss which the optimizer aims to minimize.

Post compilation, the network is fit. Fitting requires the training data to be specified, both the input and target sequence of the partitioned training dataset. The model is trained using a back-propagation algorithm (refer Section 2.4.2) for a specified number of epochs and optimized by the optimization algorithm and loss function. Batch size is specified that controls the number of training samples a network is exposed to before the weights are updated within an epoch.

The network later is evaluated on training and validation data using metrics such as the loss and accuracy of its prediction. The loss and accuracy plots of training and validation data determine the model fitness, if it’s a good fit, underfit, or overfit. Once the performance of the fit model is satisfied (good fit), the model is used to make predictions on the testing dataset.

The LSTM prediction values or forecast errors are further passed to the unsupervised ensemble learning algorithms to detect the anomalies which is described in the next section.

6.2.4.3 Anomaly Detection Process

The predictions made by the LSTM model is compared with the actual value of the input feature to calculate the forecasting errors. The forecasting errors are modeled to fit a multivariate gaussian distribution. The error vectors far away from the mean of the gaussian distribution are likely to be anomalous. Hence, a distance measure is used to compute the distance of every error vector from the distribution. Mahalanobis' distance measure is used in this approach which is an effective multivariate distance metric that computes the distance between a point and a distribution. Larger distance value indicates that the error vector point is far away from the gaussian distribution indicating that the corresponding data point as an anomaly as shown in Figure 6.11.

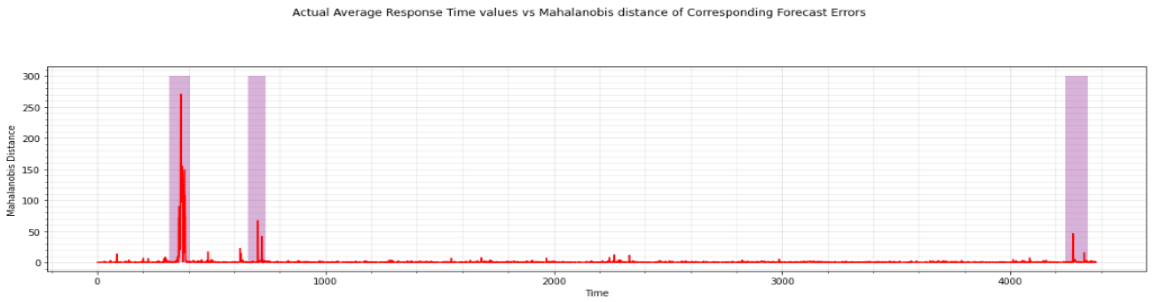


Figure 6.11: Mahalanobis distance values (y-axis) of error points from the gaussian distribution plotted against timestamp (x-axis)

These distance values can be used as anomaly scores wherein a threshold needs to be set for the distance factor in order to detect the anomalies. But this step of static thresholding is not advisable since it would require a change in threshold value when the nature of data changes. It would require the operations team to update 'n' different sets of threshold values for 'n' different services on a timely basis, which makes it no different than traditional statistical methods from the manual efforts' perspective.

To tackle this, a dynamic anomaly scoring mechanism has been used. The distance values computed by Mahalanobis' distance is fed to an unsupervised outlier detection algorithm that detects outlying distance values. The detected distance outliers correspond to the data

points that are anomalous. The distance values are fed to three different unsupervised outlier detection algorithms – (i) Isolation Forest, (ii) One-Class SVM, and (iii) LSCP – Locally Selective Combination of Parallel Outlier Ensembles. LSCP in itself is a parallel outlier ensemble that facilitates us to combine the existing unsupervised algorithms such as KNN, LOF, Isolation forest, and selects base detectors for test instance in the local region [91].

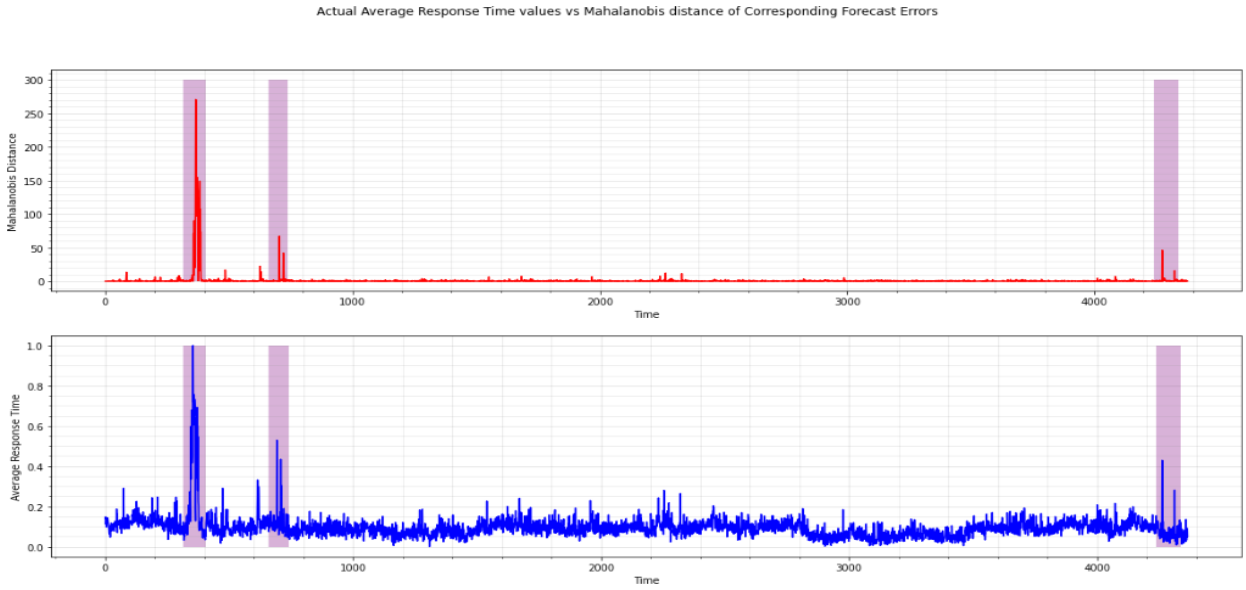


Figure 6.12: The bottom plot shows the Actual Average Response Time values of GraphQL. The top plot shows the Mahalanobis distance values of prediction error points from its distribution

The distance values of each error point from its distribution are fed to these unsupervised techniques to identify the outlying error points (points lie in the purple region of Figure 6.12 top plot) which are far away from their distribution. These outlying points indicate anomalies and thus, the data points corresponding to these outlying error points are declared as anomalies (points within the purple region of Figure 6.12 bottom plot).

6.2.5 Localization Module

The localization module aims to identify the root cause, or the causal component of the anomalies detected by the previous module, Detector module. For the detection module, the data is considered specific to the GraphQL service as mentioned in Section 6.2.4 (also

see Figure 6.3). The detection module uses the proposed LSTM and unsupervised learning ensemble to detect anomalies when GraphQL's average response time deviates from the expected baseline.

It is known that the total time taken by the GraphQL depends on the time taken by its spans/sub-components. Hence, this module determines which of its sub-component is responsible for the anomalies, i.e., responsible for degrading the overall GraphQL performance. Hence, for the localization module, the request propagated from GraphQL service to its sub-components (its child nodes) – POST, GET, CACHE is taken into account.

The total number of requests and the total time taken by each span in the transaction are extracted and grouped into 3 common buckets – POST, GET and CACHE using their name field from the tracing events data to get the individual summary information for each sub-components/buckets during the data extraction process as discussed earlier (see Figure 6.3). This individual summary information, for 3 sub-components is pre-processed in the data pre-processing module before it is fed to the detectors inside the Localization Module (see Figure 6.3).

The Localization module includes 3 different detectors for each of the 3 sub-components and a mapper which receives the results of 3 individual detectors and the GraphQL Detector module. The Detection module detects anomalies for GraphQL's average response time data, whereas the 3 individual detectors detect anomalies in case of deviation from the expected average response time for each child nodes: POST average response time, GET average response time and Cache average response time respectively. Later, the mapper uses a simple mapping technique to map the GraphQL detection results with its corresponding span/sub-components' detection results using their timestamps.

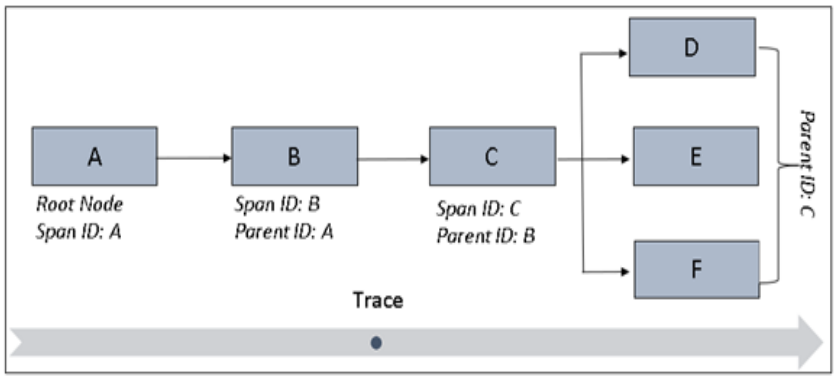


Figure 6.13: Tracing events of requests propagating from component A to C.

As depicted in Figure 6.13, consider component C as the GraphQL component and its sub-components D, E, and F as POST, GET, and Cache respectively. The mapper maps the corresponding timestamps of the GraphQL anomaly detection results (i.e., the anomalous intervals of GraphQL data) with the timestamps of its span's anomaly detection results (the anomalous intervals of POST, GET, Cache data) and declares a sub-component as the causal component when there is a match between the timestamps of their results. The mapper generates the results in the form of a report depicting the duration/timestamp of anomaly along with the details of its causal components. These results are further sent to the alert management system to take further action.

6.2.6 Information Module

The information module allows the user to access details of the tracing events (refer Figure 6.3). When a user wants to access information about a specific transaction, then they can use this module for accessing the event details. The user is presented with different options such as Trace Graph View, Trace JSON display, Summary of Trace.

i) Trace Graph View: generates a Directed Acyclic Graph (DAG) of a particular transaction or trace ID, that a user is looking for, along with the details of the parent-child relationship between the spans. It generates two different layouts of DAG – Spring layout and Kamada Kawai layout. Spring layout generates a directed graph in a spherical manner whereas Kamada Kawai layout places nodes at hierarchical levels, as shown in Figure 6.14. The directed acyclic graph displays the nodes or

spans starting from Datalayer(cyan colored node) followed by GraphQL(red node) with requests later propagating to POST (blue node), GET(green node), Cache (purple node) components.

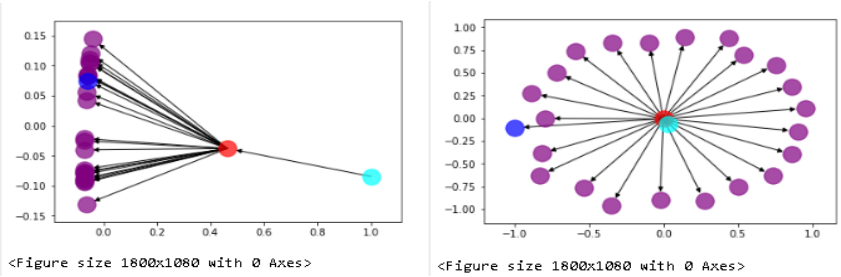


Figure 6.14: Spring layout (top plot) and Kamada Kawai layout (bottom plot)

ii) Trace JSON display: generates the trace event information in JSON format the same as how it is stored in the Elasticsearch system.

iii) Summary of Trace: This option provides a summary of the trace ID or transaction which includes the total number of spans or nodes in the transaction, Node ID, and time taken by each node in milliseconds and how each node is interconnected to each other by determining its parent-child relationship.

Chapter 7

7 Experiments and Results

The proposed approach for anomaly detection using a novel combination of LSTM deep learning model and unsupervised ensemble outlier technique - LSCP, is experimented on a real-world tracing dataset collected from running microservices that was provided by the collaborating organization. The data was collected for a duration of approximately 6 months from June 2019 to December 2019. As discussed in the previous chapter, the tracing events were grouped into 5-minutes interval, processed and key performance indicator such as Average response time and other features were used for building a detector model to forecast the future average response time of service and further predict anomalies using forecast and actual data using the proposed approach. In this section we discuss the conducted experiments, present their results and findings.

7.1 Program Libraries

The proposed approach was implemented in the Python programming language using various libraries as described in Table 6.1. The experiments were conducted on a personal computer with following specifications: Processor Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz, 2712 Mhz, 2 Core(s), 4 Logical Processor(s),8GB RAM, 1TB HDD and also on Google Colaboratory (Colab) that runs on Google Cloud server providing GPU.

Library Name	**Purpose**	**Description**
pandas ▪pandas.io.json: json_normalize ▪DataFrame ▪read_csv ▪concat ▪pandas.tseries.holiday: USFederalHolidayCalendar	Pre-processing	pandas offer data structures and operations for manipulating numerical tables and time series. It is free software released under the three-clause BSD license.
numpy	Pre-processing	NumPy is adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays.

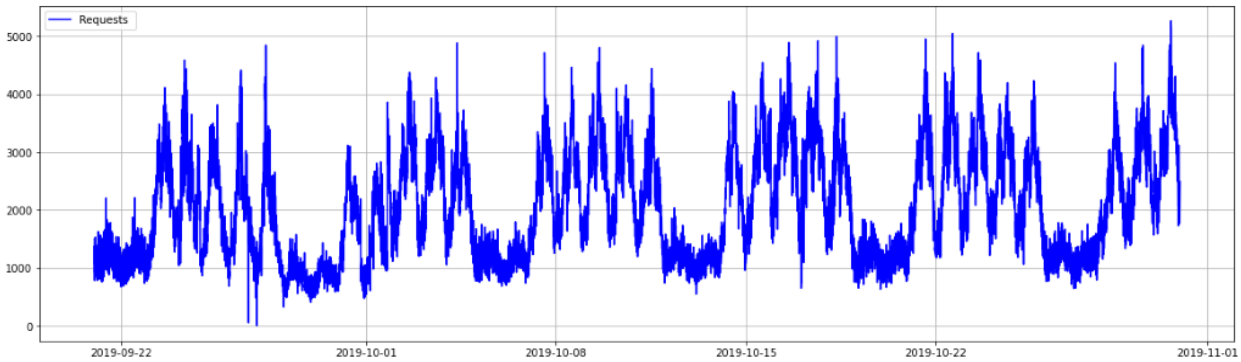
sklearn ▪preprocessing: MinMaxScaler, StandardScaler, LabelEncoder ▪sklearn.metrics: mean_squared_error, confusion_matrix, accuracy_score, classification_report, accuracy_score, precision_score, recall_score, f1_score, roc_auc_score, confusion_matrix ▪sklearn.cluster: KMeans ▪model_selection: train_test_split ▪sklearn.ensemble: IsolationForest ▪sklearn.svm: OneClassSVM	Evaluation, Data Modelling	Scikit-learn is a free software machine learning library for the Python programming language. It features various classification, regression, and clustering algorithms. It also includes matrices and preprocessing operations for the dataset
json	Data Handling	JSON (JavaScript Object Notation) encoder and decoder
urllib3	Internet Protocols and Support	urllib3 is a powerful HTTP client for Python
requests	Internet Protocols and Support	Requests verify SSL certificates for HTTPS requests
certifi	Internet Protocols and Support	Certifi is a selected collection of Root Certificates, extracted from the Requests project, to validate the trustworthiness of SSL certificates verify TLS hosts' identity
smtplib ▪ SMTPException	Internet Protocols and Support	Used to send mail to any Internet machine with an SMTP or ESMTP listener daemon by defining an SMTP client session object.
pickle	Data Persistence	Used to serialize and de-serialize Python object structure
elasticsearch	Data Access Protocol	Used to access ElasticSearch system using python
time	Time access and Conversions	Provides various time-related functions
datetime ▪ datetime, date, timedelta	Data Types	Provides classes to manipulate dates and times
dateutil.parser	Data Types	Used to parse most known formats to represent a date and/or time
pydrive ▪pydrive.auth ▪pydrive.drive	Internet Protocols and Support	PyDrive is a wrapper library of google-api-python-client used to simplify Google Drive API tasks
google.colab	Protocol	Colab is cloud service offered by Google to run notebooks on its server. It offers GPU for free.

os	Generic Operating System Services	Provides a portable way of using operating system dependent functionality such as reading a file
matplotlib ▪pyplot: plot, show	Data Visualization	To create interactive visualizations in Python
keras ▪models: Sequential ▪layers: LSTM, Dense, Dropout ▪callbacks: EarlyStopping ▪optimizers	Neural Network	Keras is an open-source library or high-level neural networks APIs, written in Python and supports multiple back-end neural network computation engines (TensorFlow, Microsoft Cognitive Toolkit, R, Theano, or PlaidML).
networkx	Directed Acyclic Graph Visualization	NetworkX is a Python package used to create, manipulate complex graphs and networks.
pyod ▪ pyod.models.iforest: IForest ▪ pyod.models.knn: KNN ▪ pyod.models.lof: LOF ▪ pyod.models.lscp: LSCP	Data Modelling	pyod is python toolkit for outlier detection and provides various individual algorithms, Outlier Ensemble and Detector Combination Frameworks

Table 7.1: List of Libraries used

7.2 Exploratory Data Analysis

After the data is pre-processed, it is examined to check the patterns in it visually through graphical representations. A subset of data is visualized graphically in order to understand the behavior and nature of data such as the request arrival pattern during weekdays and weekends as shown in Figure 7.1.a) and processing time for those requests as shown in Figure 7.1.b).



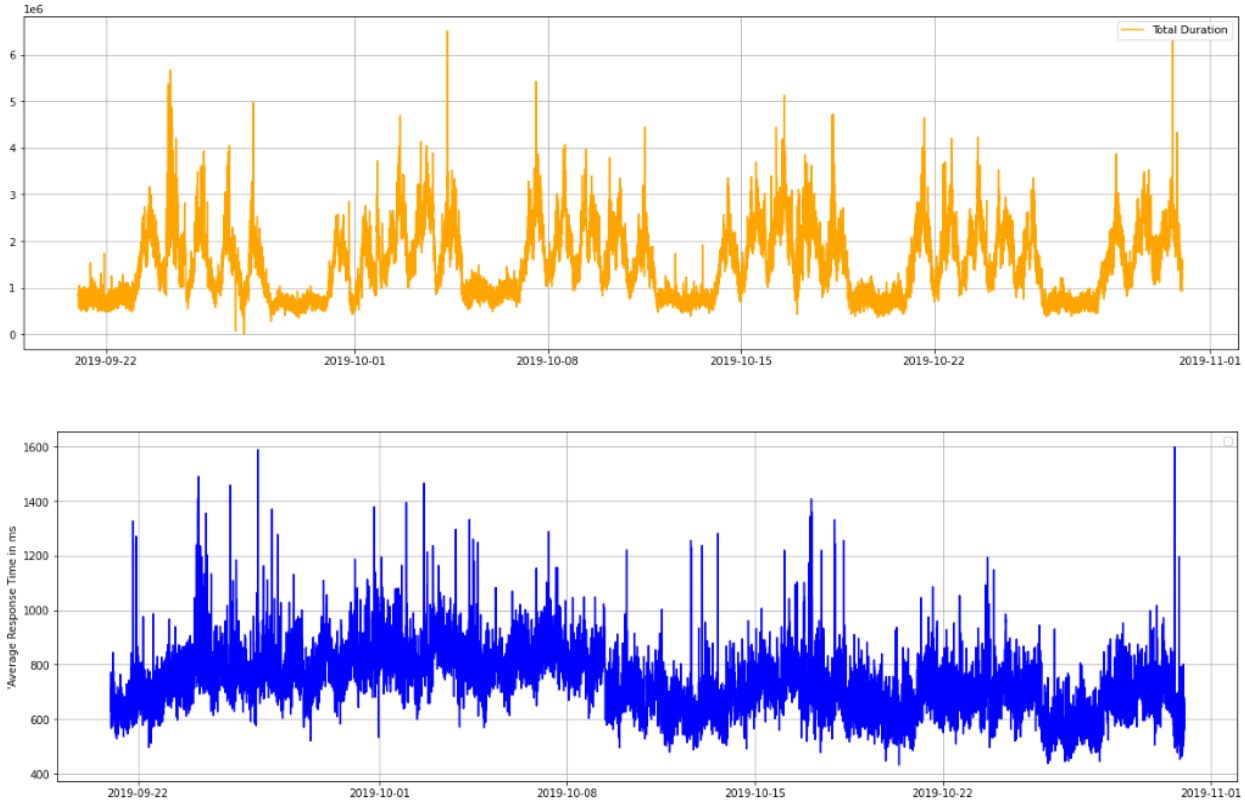


Figure 7.1: a) Total number of Requests for every minute interval captured for over a month (top plot), b) Total Duration taken by GraphQL service to process those requests (middle plot), c) Average Response time of GraphQL service for the total number of requests and its processing time (bottom plot).

As shown in Figure 7.1.a), we observe the pattern of the total number of requests that the GraphQL service receives. It follows a periodic seasonality or repeated patterns (weekly). There are fewer requests during the weekends (such as the lower spikes on 21st & 22nd September 2019) compared to the weekdays. And for a given day, there is a large number of requests during the daytime compared to the night as shown in Figure 7.2.a). The number of requests fluctuates with an increasing trend throughout the day reaching a peak around the afternoon which continues to decrease further towards the end of the day.

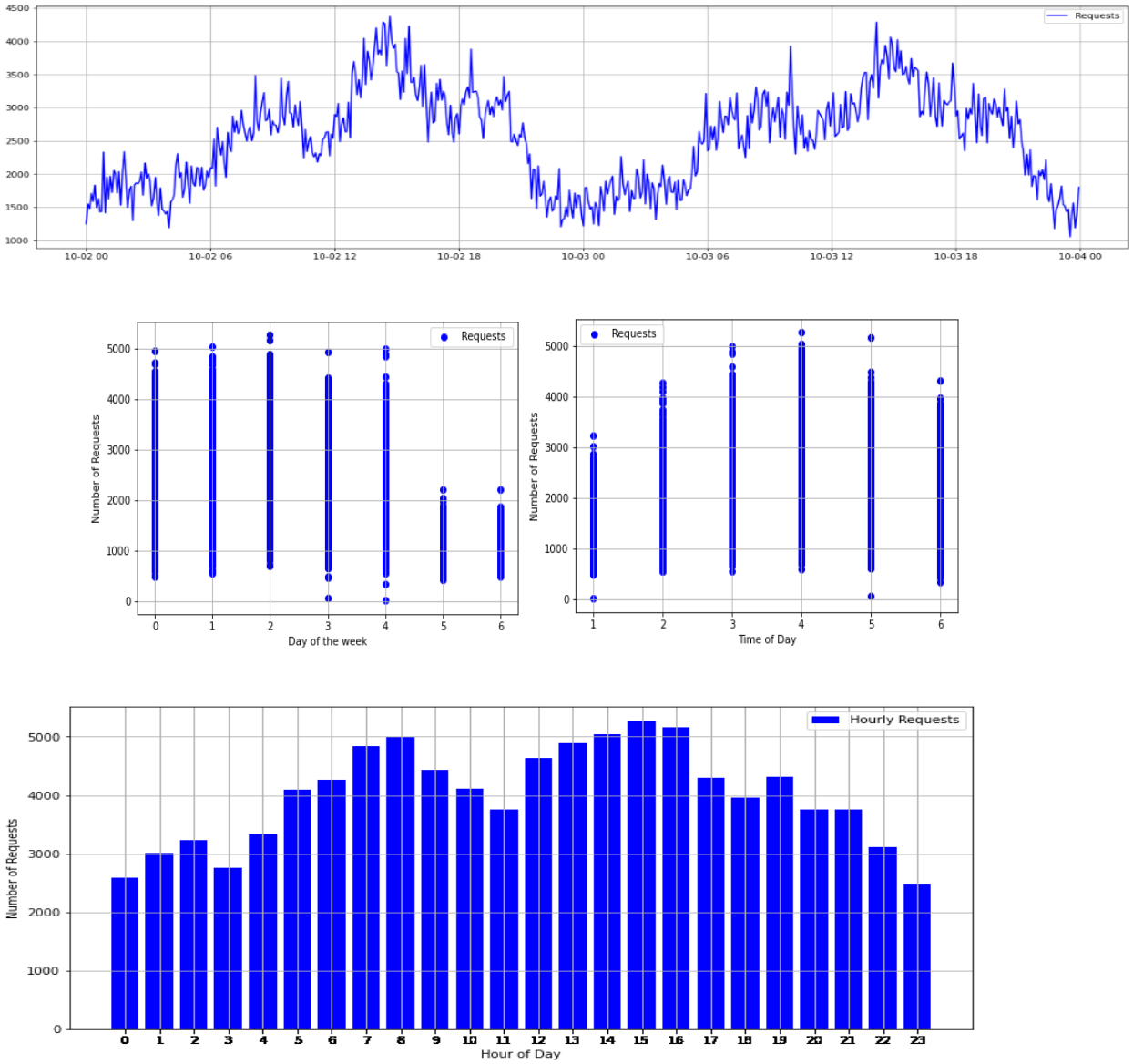


Figure 7.2: a) Total Number of Requests in a day (top plot), b) Total Number of Requests for a day of the week (leftmost middle plot), c) Total Number of Requests for a time of the day (rightmost middle plot), d) Total Number of Requests for an hour of the week (bottom plot)

The number of requests is plotted against the day of the week where '0' indicates Monday, '1' as Tuesday and so on, as shown in Figure 7.2.b) depicting a low number of requests during weekends. In Figure 7.2.c), the total number of requests feature is plotted against 'time of the day' where values (1-6) are denoted as follows:

1 - *Midnight: 12 am to 4 am*
2 - *Early morning: 4 am-7 am*

3 - *Morning: 7 am-12 pm*
4 - *Afternoon: 12 pm-4 pm*
5 - *Evening: 4 pm-7 pm*
6 - *Night: 7 pm-12 am*

As per Figure 7.2.c), we observe that GraphQL receives a large number of requests during the afternoon between 12 pm to 4 pm and least during midnight from 12 am to 4 am, which can also be seen in Figure 7.2.d). Once, the data is explored visually to understand the pattern, the next step is to accordingly use the data in an appropriate manner to build the model. Data Analysis is crucial to understand how to split the series data into input and target sequences, and its consequences while training the LSTM time series model. The *lookback* and *future_steps* parameter provided to the model is highly dependent on the nature of the dataset, hence it is visually explored and analyzed beforehand to understand its temporal behaviour.

7.3 Experiments on Detection Module

The summary data (Total Requests, Total Duration) generated from the Extraction module is pre-processed in the Data Pre-processing module, which generates new features (Average Response time, weekday/weekend, day of the week, etc.) and normalizes the data before it is fed to the Detection module. The dataset consisted of 53,410 samples from June 2019 till December 2019. By samples, we mean the traces or transactions processed by the GraphQL layer. For the first set of experiments on Anomaly detection, the last half portion of the data (from 21st September 2019) was considered. We conducted two main experiments by modeling two variants of LSTM: Univariate LSTM and Multivariate LSTM. Each of these two models was tested for different prediction lengths: Single-Step and Multi-Step. The details for these experiments are described as follows:

7.3.1 Experiment 1: Univariate LSTM + Unsupervised Ensemble

For experiment 1, univariate data is used to build the LSTM model. In univariate, only one feature is considered, which in our case was ‘Average Response Time’, ordered in a timely fashion using timestamp feature. The dataset used for the experiment is from 21st September to 30th October 2019, resulting in 11521 samples, which is divided into a 62:38 ratio for training and testing purposes. The data used for training is assumed to not have

any sort of anomalies and thus is selectively taken from 21st September until 15th October 2019 and the remaining 38% until 30th October is used for testing the model. Each of the training and testing samples is converted into respective formats of input and target sequences required by the LSTM time-series model using the parameters – '*lookback'* and '*future_steps'*. The model uses '*lookback'* number of time steps as its input to predict '*future_steps'* number of time steps. The model is tested for various combinations of input length and prediction length. Input length or a '*lookback*' of 12, 24, 48, 288 is chosen, which for a 5-minute interval of data represents the usage of the previous 1 hour, 2 hours, 4 hours, 24 hours values as lookbacks respectively.

7.3.1.1 Experiment 1.1 - Single Step

For experiment 1.1, lookback of 12 and prediction length of 1 was used. An LSTM model with 5 recurrent layers in total with a decreasing number of neurons [64, 48, 32, 16, 8] from top to bottom layer followed by a dense output layer with the number of neurons the same as prediction length(*future_steps*) was built as given in Table 7.2. The first layer with 64 neurons accepts input samples with a *lookback* of 12 timesteps and its output is fed to the next recurrent layer after a drop out of 20%. Each recurrent layer is followed by a drop out layer which stochastically reduces the number of neurons while training to prevent overfitting of the model. The prediction model was trained using Adam optimizer with a learning rate of 0.0001 and the default values of beta_1 as 0.9, beta_2 as 0.999, epsilon as 1e-08, decay as 0.0, using Mean Squared Error as loss function and a batch size of 30. The model was trained for 100 epochs and during each epoch, 20% of the training samples were used for validation purposes. The trained model is evaluated using test data as shown in Figure 7.3.

Experiments	Model Architecture	Optimizer	Epochs	Batch Size	Lookback	Future Steps	RMSE
Experiment 1	LSTM (64) Dropout (0.2) LSTM (48) Dropout (0.2) LSTM (32) Dropout (0.2) LSTM (16) Dropout (0.2) LSTM (8) Dense	Adam: learning rate=0.0001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	100	30	12	1	Train Score: 88.63 RMSE Test Score: 145.47 RMSE

Table 7.2: LSTM Model details for lookback 12 and prediction length 1

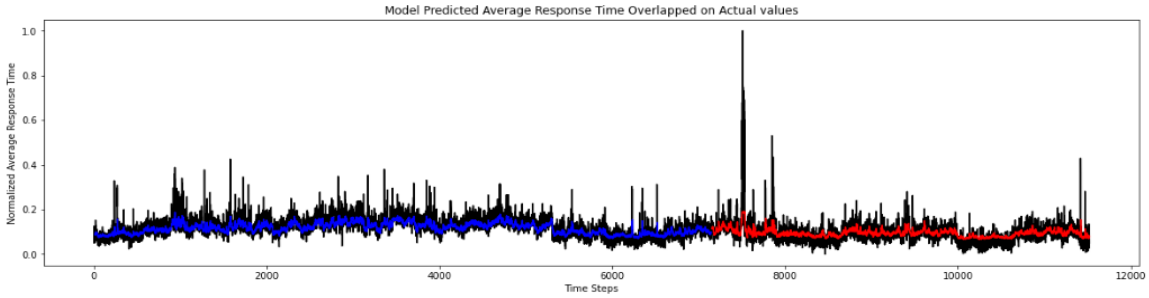


Figure 7.3: LSTM Training and Prediction. The model predicted average response time values for the training set (in blue) and the testing set (in red) is plotted overlapping on top of actual or expected average response time values (in black).

After forecasting, the forecast errors computed using actual and predicted values were modeled using gaussian distribution. Mahalanobis distance measure was used to calculate the distances of each error vector point from its distribution using mean and covariance variables of the errors. The error points which are far away from the distribution indicates that those points as anomalies.

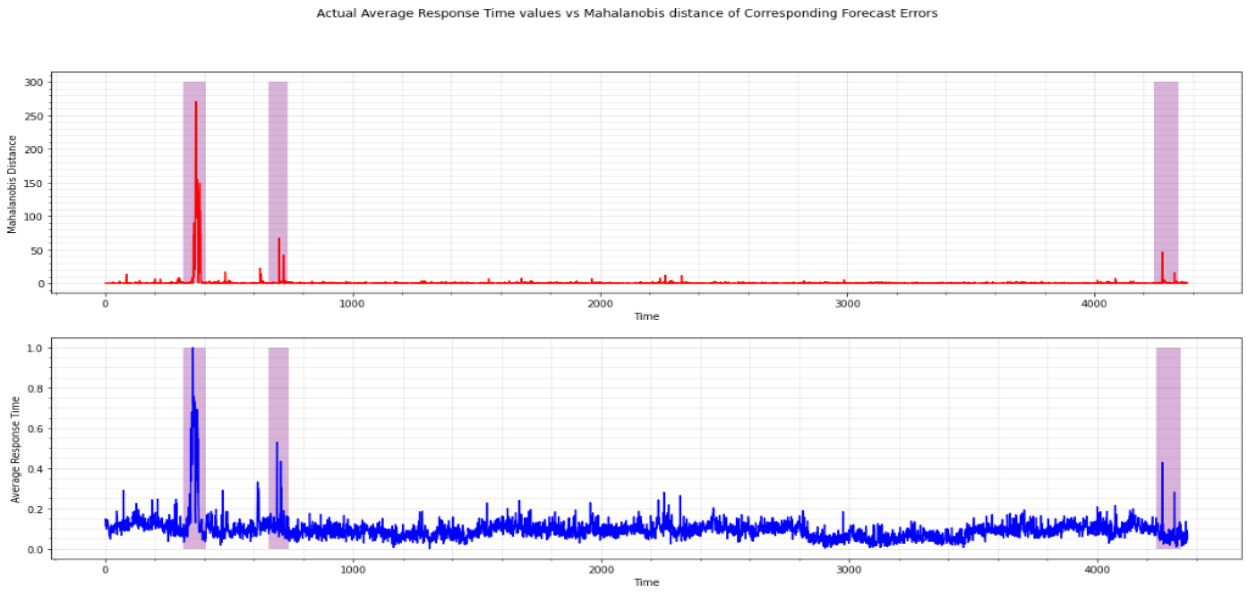


Figure 7.4: The bottom plot shows the Actual Average Response Time values (testing set). The top plot shows the Mahalanobis distance values of prediction error points from its distribution.

As shown in Figure 7.4, the points with higher Mahalanobis distance values, shaded in purple, indicates the existence of anomalies in that region. This is plotted against actual

values where we observe that the shaded region in the bottom plot at the same spots as the corresponding top plot shows higher average response time values, which are nothing but the anomalies. Hence, a higher Mahalanobis distance value indicates a larger prediction error which in turn reflects the corresponding data point as an anomaly.

To identify the outlying Mahalanobis distance (*m_dist*) values for anomaly detection, these distance values were fed to 3 different types of unsupervised learning techniques: Isolation Forest, One-Class SVM, and LSCP methods. Apart from the proposed approach, an existing threshold-based approach was experimented using two different values of threshold (5, 10) to detect the anomalies and compare its results with the proposed approaches.

For the proposed approaches, the following hyper-parameters were used as follows. An Outlier fraction of 0.01 was used for all the 3 techniques to check which error points were far away from the Gaussian distribution, i.e., outlying Mahalanobis distance values. For the ‘LSTM + LSCP’ ensemble method, 4 different variations of LSCP ensemble methods were used. In ‘LSCP-1’ ensemble method, LOF with 10 neighbors and 0.01 contamination as parameters, and KNN with default parameters were used as base detectors. In the ‘LSCP-2’ ensemble, three LOFs each with neighbors 12, 24, 48, and 0.01 contamination parameters respectively were used as base detectors along with KNN. In the ‘LSCP-3’ ensemble method, four LOFs each with neighbors 12, 24, 48, 60, and 0.01 contamination as parameters respectively were used along with KNN. In the ‘LSCP-4’ ensemble, the outlier fraction was set to 0.009 instead of 0.01, unlike previous ensembles and four LOFs with neighbors 12, 24, 36, 48, and 0.01 contamination along with KNN using neighbors 12 as parameters were used.

Comparison of results by ‘LSTM + Isolation Forest’, ‘LSTM + One-Class SVM’, ‘LSTM + LSCP’ and ‘LSTM + Existing Threshold’ approaches are shown below.

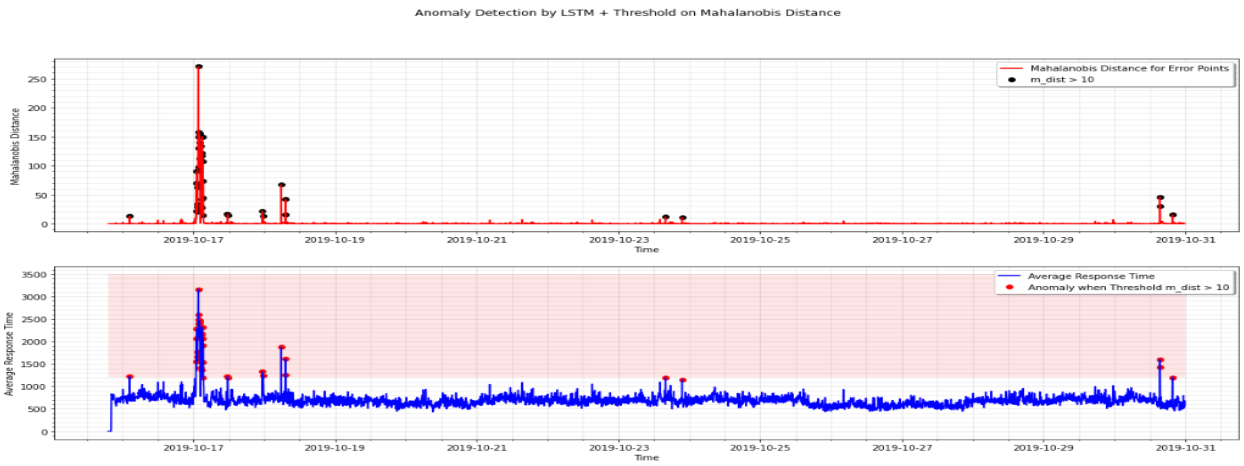


Figure 7.5: Anomalies detected using a threshold of Mahalanobis distance > 10.

In Figure 7.5, the black dots in the top plot represents the Mahalanobis distance values of prediction errors greater than the cut-off value of 10. The bottom plot shows the actual 'average response time' values (testing set) with anomalies (red dots) detected when a threshold of *m_dist* or Mahalanobis distance greater than 10 is applied. Similarly, the black dots in the top plot of Figures 7.6 and 7.7 denote the outlying distance values of forecast errors detected by LSCP-4 and OC-SVM respectively. Their bottom plot shows the actual average response time values with anomalies detected when its corresponding *m_dist* values were detected as outliers by LSCP-4 and OC-SVM respectively.

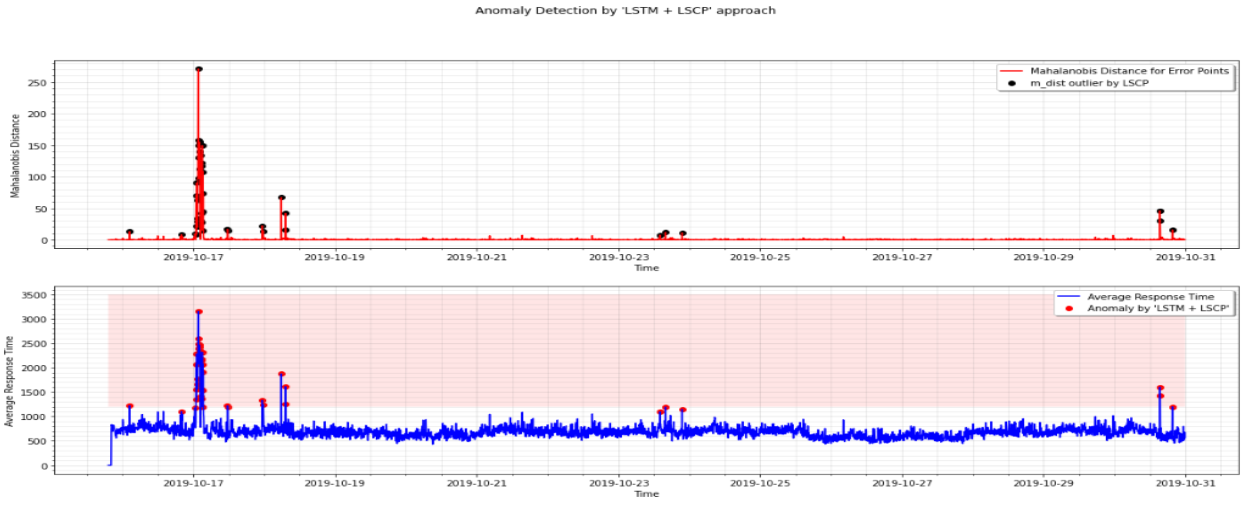


Figure 7.6: Anomalies detected using LSTM + LSCP-4 ensemble approach

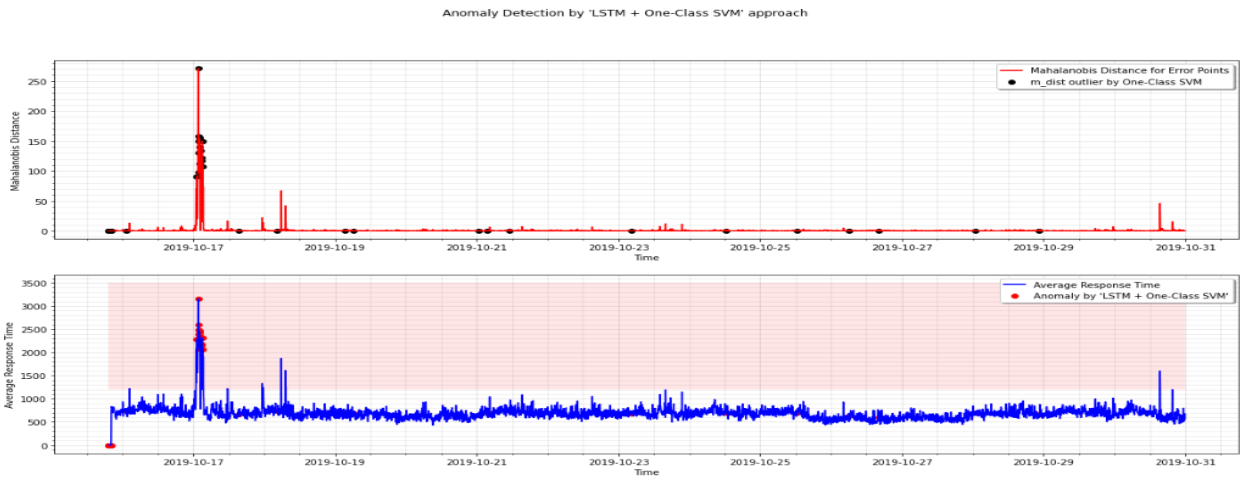


Figure 7.7: Anomalies detected using LSTM + One-Class SVM ensemble approach.

The data points or peaks falling in the red shaded region of the bottom plots are considered as anomalies. These individual data points detected as anomalies can be termed as point anomalies. Also, since we group the requests into a 5-minute interval, these anomalies fall under the category of collective anomalies as well. This region has been identified as the anomaly zone area by the validation experts from the collaborating organization for detecting and alerting the points falling in this region as anomalies.

We have detected anomalies for the testing data and captured their results for all the mentioned ensemble approaches as given in Table 7.3 and Table 7.4 which is discussed below. As per the anomaly zone identified by the domain experts, we labeled the test data manually under their supervision.

Apart from validating the results visually through graphs for all the individual dates of test data, the manually labeled data was used to generate a confusion matrix for each of the tested approaches and metrics such as precision, recall, F1-score, specificity, and accuracy were calculated. In unsupervised learning, where the data is unlabeled, the results are not validated by such techniques and are not appropriate either. But we leveraged the label information to validate the unsupervised methods like any other supervised method under the guidance of domain experts.

Confusion Matrix: [[TN FP] [FN TP]]

LSTM + Threshold 5	LSTM + Threshold 10	LSTM + LSCP1	LSTM + LSCP2	LSTM + LSCP3	LSTM + LSCP4	LSTM + Isolation Forest	LSTM + OCSVM
Confusion Matrix : [[4322 20] [0 36]]	Confusion Matrix : [[4337 5] [1 35]]	Confusion Matrix : [[4333 9] [1 35]]	Confusion Matrix : [[4328 14] [6 30]]	Confusion Matrix : [[4333 9] [1 35]]	Confusion Matrix : [[4337 5] [1 35]]	Confusion Matrix : [[4335 7] [0 36]]	Confusion Matrix : [[4317 25] [22 14]]

Table 7.3: Experiment – 1.1 Results: Confusion Matrix of all the tested approaches

Method	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.643	1.000	0.995	0.783	0.995	20 + 0 = 20
LSTM + Threshold 10	0.875	0.972	0.999	0.921	0.999	5 + 1 = 6
LSTM + LSCP1	0.795	0.972	0.998	0.875	0.998	9 + 1 = 10
LSTM + LSCP2	0.682	0.833	0.997	0.750	0.995	14 + 6 = 20
LSTM + LSCP3	0.795	0.972	0.998	0.875	0.998	9 + 1 = 10
LSTM + LSCP4	0.875	0.972	0.999	0.921	0.999	5 + 1 = 6
LSTM + Isolation Forest	0.837	1.000	0.998	0.911	0.998	7 + 0 = 7
LSTM + One-Class SVM	0.359	0.389	0.994	0.373	0.989	25 + 22 = 47

Table 7.4: Experiment-1.1 Results: Using Evaluation Metrics

Given the fact that in an anomaly detection problem, there is a large proportion of normal points and very few outliers in the dataset, we consider the metrics such as precision, recall, and F1-score rather than the accuracy metric to evaluate the models for such imbalanced datasets. As per the results tabulated in Table 7.4, 'LSTM + LSCP-4' and 'LSTM + Threshold10' has achieved a higher precision score of 87.5%, followed by LSTM + Isolation Forest' with a precision score 83.7% respectively. In terms of recall metric, 'LSTM + Isolation Forest' and 'LSTM+Threshold-5' has achieved 100%, followed by 'LSTM + LSCP-1', 'LSTM + LSCP-3', 'LSTM + LSCP-4' and 'LSTM + Threshold-10' all with 97.2% respectively.

- 87.5% Precision ($\sim \frac{7}{8}$ fraction) signifies that out of every 8 anomalies detected by the 'model', 7 of them are identified correctly by the model.
- 100% Recall signifies that the model identifies all the anomalies present in the 'system'. If there are 10 anomalies in the system, the model identifies all the10.

Since it is important to 'correctly' identify anomalies (Precision) and also to identify 'all' anomalies (Recall), we calculate the F1-score which gives equal weightage to both precision and recall in order to compare the performance of different models.

As shown in Table 7.4, ‘LSTM + LSCP-4’ and ‘LSTM + Threshold-10’ has a higher F1-score of 92.1% followed by ‘LSTM + Isolation Forest’ and LSTM + LSCP-3’ with F1-scores of 91.1% and 87.5% respectively. Among all the ensembles, ‘LSTM+OC-SVM’ has the lowest F1-score of 37.3%.

As per the F1-Score measure, with 92.1% score, the ‘LSTM + LSCP-4’ and ‘LSTM + Threshold-10’ models are the best models for this experiment.

Dual Validation - To reiterate, the dataset was split into training and testing set to train the LSTM model to learn the patterns in the data, further predictions were made using the test set. The prediction errors/distance values of error points were fed to the second module of unsupervised outlier detection methods (LSCP/OC-SVM/IF) to detect the anomalies present in the test set. Here, the unsupervised techniques in the second module were directly fit and tested on the same test set. Hence, to evaluate the generality of the second module, the entire combination of LSTM and unsupervised outlier detection ensemble was evaluated as a whole for another sample of data. The data tested previously was a subset of historical data. Hence, for the secondary validation of the whole proposed combination, a very small sample of data from 16th to 18th December 2019 shown in Figure 7.8, was used for evaluation.

This data was also extracted in the same way as the previous dataset from September to October for every 5-minutes interval. The dataset was chosen for specific dates, which are around 2 months ahead of the previous dataset, to test the generalization and applicability of the proposed ensemble detection model throughout months without requiring re-training. Secondly, the length of data is cut short to 2 days rather than using a large dataset so as to check the capability of the proposed approach on short term data as opposed to a historical large set. For the ‘LSTM + LSCP-4’ method used in experiment 1.1, a dual validation was conducted wherein the ‘LSCP-4' in the second module got evaluated for unseen data, thus validating the whole ensemble, ‘LSTM + LSCP-4’, a second time. The results are captured for this data as given in Figure 7.9.

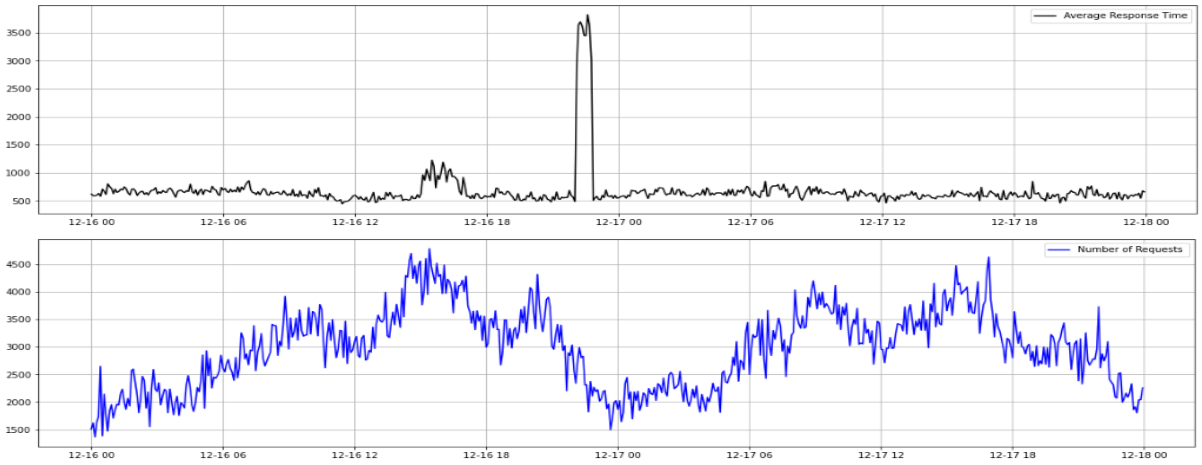


Figure 7.8: Average Response Time and Total number of Requests for GraphQL from 16th - 18th December 2019.

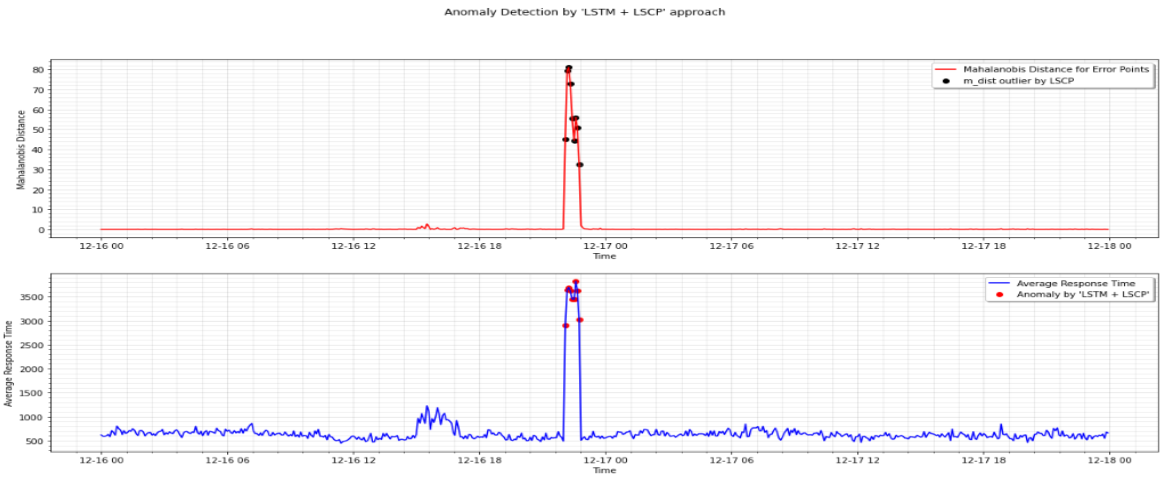


Figure 7.9: Anomalies Detected by the 'LSTM + LSCP-4' method for data from 16th - 18th December 2019.

7.3.1.2 Experiment 1.2 - Multi-Step

For experiment 1.2, the prediction lengths used is more than 1, i.e., a value of multiple time steps ahead is predicted rather than a single time step. Lookback and prediction lengths of (lookback, future steps) = (12,12), (24,24) and (48,48) are used for multi-step models. The LSTM model had 5 recurrent layers in total with a decreasing number of neurons [512, 256, 128, 64, 32] from top to bottom followed by a dense output layer with the number of neurons the same as prediction length (*future_steps*). A drop out of 20% is used after each recurrent layer. For experiments under multistep predictions, Adam optimizer with

different learning rates was used as given in Table 7.5, and the default values of beta_1 as 0.9, beta_2 as 0.999, epsilon as 1e-08, decay as 0.0, Mean Squared Error as loss function and a batch size of 1024 were used. The model was trained for 200 epochs and during each epoch, 20% of the training samples were used for validation purposes. The detection procedure has the same steps as mentioned in the previous experiment 1.1. The forecast errors are fed to the second module with different unsupervised outlier detection approaches forming ensembles: 'LSTM + Isolation Forest', 'LSTM + One-Class SVM', 'LSTM + LSCP' and 'LSTM + Static Threshold'.

Experiments	Model Architecture	Optimizer	Epochs	Batch Size	Lookback	Future Steps	RMSE
Experiment 1.2.1	LSTM (512) Dropout (0.2) LSTM (256) Dropout (0.2) LSTM (128) Dropout (0.2) LSTM (64) Dropout (0.2) LSTM (32) Dense	Adam: learning rate=0.001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	200	1024	12	12	Train Score: 84.61 RMSE Test Score: 131.60 RMSE
Experiment 1.2.2	LSTM (512) Dropout (0.2) LSTM (256) Dropout (0.2) LSTM (128) Dropout (0.2) LSTM (64) Dropout (0.2) LSTM (32) Dense	Adam: learning rate=0.0001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	200	1024	24	24	Train Score: 86.28 RMSE Test Score: 121.74 RMSE
Experiment 1.2.3	LSTM (512) Dropout (0.2) LSTM (256) Dropout (0.2) LSTM (128) Dropout (0.2) LSTM (64) Dropout (0.2) LSTM (32) Dense	Adam: learning rate=0.0001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	200	1024	48	48	Train Score: 88.98 RMSE Test Score: 134.74 RMSE

Table 7.5: LSTM Model details for Multistep univariate data

For experiments 1.2.1, 1.2.2, and 1.2.3, in LSCP-1 ensemble method, LOF with 10 neighbors and 0.01 contamination, and KNN with default parameters were used as base detectors. In the LSCP-2 ensemble, three LOFs each with neighbors 12, 24, 48, and 0.01

contamination parameters respectively were used as base detectors along with KNN. In the LSCP-3 ensemble method, four LOFs each with neighbors 12, 24, 48, 60, and 0.01 contamination as parameters respectively were used along with KNN. In LSCP-4 ensemble, outlier fraction was set to 0.009 instead of 0.01 unlike previous ensembles as a measure of tuning the hyperparameter to reduce false positives and false negatives, and four LOFs with neighbors 12, 24, 36, 48, 60 with 0.01 contamination, and KNN using neighbors 12 as parameters were used.

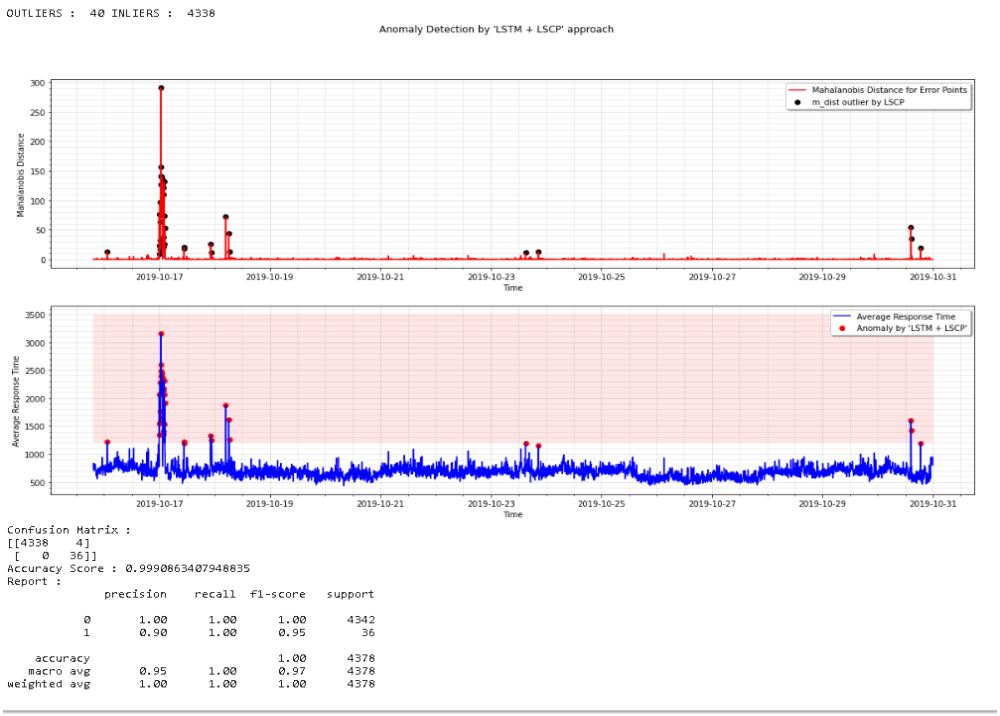


Figure 7.10: Experiment 1.2.1 Detection results for 'LSTM+LSCP-4'ensemble

Experiments	LSTM + Threshold 5	LSTM + Threshold 10	LSTM + LSCP1	LSTM + LSCP2	LSTM + LSCP3	LSTM + LSCP4	LSTM + Isolation Forest	LSTM + OCSVM
Experiment 1.2.1	Confusion Matrix : [[4318 24] [0 36]]	Confusion Matrix : [[4337 5] [1 35]]	Confusion Matrix : [[4316 26] [18 18]]	Confusion Matrix : [[4330 12] [4 32]]	Confusion Matrix : [[4334 8] [0 36]]	Confusion Matrix : [[4334 8] [0 36]]	Confusion Matrix : [[4334 8] [0 36]]	Confusion Matrix : [[4315 27] [22 14]]
Experiment 1.2.2	Confusion Matrix : [[4297 45] [4 32]]	Confusion Matrix : [[4320 22] [4 32]]	Confusion Matrix : [[4316 26] [18 18]]	Confusion Matrix : [[4321 21] [13 23]]	Confusion Matrix : [[4322 20] [12 24]]	Confusion Matrix : [[4331 11] [7 29]]	Confusion Matrix : [[4328 14] [6 30]]	Confusion Matrix : [[4310 32] [23 13]]
Experiment 1.2.3	Confusion Matrix : [[4302 40] [2 34]]	Confusion Matrix : [[4331 11] [4 32]]	Confusion Matrix : [[4318 24] [16 20]]	Confusion Matrix : [[4328 14] [6 30]]	Confusion Matrix : [[4328 14] [6 30]]	Confusion Matrix : [[4331 11] [3 33]]	Confusion Matrix : [[4330 12] [4 32]]	Confusion Matrix : [[4315 27] [21 15]]

Table 7.6: Experiment - 1.2 Results: Confusion Matrix of all models for all the experiments

Experiment 1.2.1 Methods	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.600	1.000	0.994	0.750	0.995	24 + 0 = 24
LSTM + Threshold 10	0.875	0.972	0.999	0.921	0.999	5 + 1 = 6
LSTM + LSCP1	0.409	0.500	0.994	0.450	0.990	26 + 18 = 44
LSTM + LSCP2	0.727	0.889	0.997	0.800	0.996	12 + 4 = 16
LSTM + LSCP3	0.818	1.000	0.998	0.900	0.998	8 + 0 = 8
LSTM + LSCP4	0.900	1.000	0.999	0.947	0.999	4 + 0 =4
LSTM + Isolation Forest	0.818	1.000	0.998	0.900	0.998	8 + 0 = 8
LSTM + One-Class SVM	0.341	0.389	0.994	0.364	0.989	27+22 = 49

Table 7.7: Experiment - 1.2.1 Results: Using Evaluation Metrics

As per the results for experiment 1.2.1 given in Table 7.8, 'LSTM + LSCP-4' has performed better than all other approaches with higher precision and recall scores of 90% and 100% respectively, resulting in F1-Score of 94.7%, highest among other tested approaches.

- 90% Precision indicates that out of 10 Anomalies detected by the "model", 9 of them are identified correctly.
- 100% Recall indicates that the model identifies all the anomalies present in the "system". If there are 10 anomalies in the system, the model identifies all the 10.

'LSTM + Threshold10' method has achieved an F1-Score of 92.1% followed by 'LSTM + Isolation forest' and 'LSTM + LSCP-3' both with an F1-Score of 90%. 'LSTM + OC-SVM' performed poorly with an F1-Score of 36.4%. With the highest F1-Score of 94.7%, LSTM+LSCP-4 is the best model for this experiment.

Experiment 1.2.2 Methods	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.416	0.889	0.990	0.566	0.989	45 + 4 = 49
LSTM + Threshold 10	0.593	0.889	0.995	0.711	0.994	22 + 4 = 26
LSTM + LSCP1	0.409	0.500	0.994	0.450	0.990	26 + 18 = 44
LSTM + LSCP2	0.523	0.639	0.995	0.575	0.992	21 + 13 = 34
LSTM + LSCP3	0.545	0.667	0.995	0.600	0.993	20 + 12 = 32
LSTM + LSCP4	0.725	0.806	0.997	0.763	0.996	11 + 7 = 18
LSTM + Isolation Forest	0.682	0.833	0.997	0.750	0.995	14 + 6 = 20
LSTM + One-Class SVM	0.289	0.361	0.993	0.321	0.987	32 + 23 = 55

Table 7.8: Experiment - 1.2.2 Results: Using Evaluation Metrics

In experiment 1.2.2, 'LSTM + LSCP-4' has achieved a higher F1-Score of 76.3% followed by 'LSTM + Isolation Forest' and 'LSTM + Threshold10' with F1-Scores of 75% and 71.1% respectively. 'LSTM + OC-SCM' and 'LSTM + Threshold-5' have performed poorly with lowest F1-Scores of 32.1% and 56.6% respectively as displayed in Table 7.8.

Experiment 1.2.3 Method	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.459	0.944	0.991	0.618	0.990	40 + 2 = 42
LSTM + Threshold 10	0.744	0.889	0.997	0.810	0.997	11 + 4 = 15
LSTM + LSCP1	0.455	0.556	0.994	0.500	0.991	24 + 16 = 40
LSTM + LSCP2	0.682	0.833	0.997	0.750	0.995	14 + 6 = 20
LSTM + LSCP3	0.682	0.833	0.997	0.750	0.995	14 + 6 = 20
LSTM + LSCP4	0.750	0.917	0.997	0.825	0.997	11 + 3 = 14
LSTM + Isolation Forest	0.727	0.889	0.997	0.800	0.996	12 + 4 = 16
LSTM + One-Class SVM	0.357	0.417	0.994	0.385	0.989	27 + 21 = 48

Table 7.9: Experiment - 1.2.3 Results: Using Evaluation Metrics

As per the results in Table 7.9 for experiment 1.2.3, 'LSTM + LSCP-4' has the highest F1-Score of 82.5% followed by 'LSTM + Threshold-10' and 'LSTM + Isolation Forest' with F1-Scores of 81% and 80% respectively. 'LSTM + OC-SCM' has the lowest F1-Score of 38.5%.

7.3.2 Experiment 2: Multivariate LSTM + Unsupervised Ensemble

For experiment 2, multivariate data is used to build the LSTM model. In contrast to univariate, where only one feature is considered ('Average Response Time') multivariate uses more than one feature for building the model. In our case, apart from the 'Average response time' feature, we used other features that were engineered out of timestamps during the data pre-processing stage. These features include the day of the week, time of the day, daylight, hours of the day, Weekday or weekend, holiday or not, average response time 5 minutes back, an hour back, and a day back from the current timestamp. Though LSTM is a time-series model in itself, capturing the temporal characteristics of the data, we wanted to experiment with how explicit inclusion of features engineered out of timestamps creates a difference in model learning and its prediction.

The dataset used for the experiment is from 21st September to 30th October 2019, resulting in 11521 samples, the same as experiment 1. The data is divided into 62:38 for training and testing the model. Both the training and testing samples were converted into respective formats of input and target sequences required by the LSTM time-series model using the parameters - *lookback* and *future_steps* inclusive of all the features. Input length or lookback of 12, 24 is chosen, and for a 5-minute interval data, these values indicate usage of previous 1 hour, 2 hours values as lookbacks respectively.

7.3.2.1 Experiment 2.1 - Single Step

For experiment 2.1, lookback of 12 and prediction length of 1 is used. The LSTM model had 7 recurrent layers in total with a decreasing number of neurons [512, 256, 128, 64, 32, 16, 8] from top to bottom followed by a dense output layer with the number of neurons same as prediction length (*future_steps = 1*). Each recurrent layer is followed by a 20 % drop out layer. We trained the prediction model using Adam optimizer with a learning rate of 0.00001, used Mean Squared Error for loss function, and a batch size of 256. The model was trained for 400 epochs and during each epoch, 20% of the training samples were used for validation. Like all previous experiments, the forecast errors are fed to different unsupervised outlier detection techniques.

Experiments	Model Architecture	Optimizer	Epochs	Batch Size	Lookback	Future Steps	RMSE
Experiment 2.1	LSTM (512) Dropout (0.2) LSTM (256) Dropout (0.2) LSTM (128) Dropout (0.2) LSTM (64) Dropout (0.2) LSTM (32) Dropout (0.2) LSTM (16) Dropout (0.2) LSTM (8) Dense	Adam: learning rate=0.00001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	400	256	12	1	Train Score: 103.86 RMSE Test Score: 147.13 RMSE

Table 7.10: LSTM Model details for Single-Step Multivariate data.

For experiment 2.1, all the unsupervised techniques with outlier fraction 0.01 were used similar to experiment 1.2.1. In the ‘LSCP-1’ ensemble method, LOF with 10 neighbors,

and KNN with default parameters were used as base detectors. In the 'LSCP-2' ensemble, three LOFs each with neighbors 12, 24, 48 were used as base detectors along with KNN. In the 'LSCP-3' ensemble method, four LOFs each with neighbors 12, 24, 48, 60 respectively were used along with KNN. In LSCP-4 ensemble, outlier fraction was set to 0.009, and four LOFs with neighbors 12, 24, 36, 48, and KNN with neighbors 12 were used in LSCP 4 method. For all the LOFs, for the above four LSCPs, contamination of 0.01 was used.

Experiments	LSTM + Threshold 5	LSTM + Threshold 10	LSTM + LSCP1	LSTM + LSCP2	LSTM + LSCP3	LSTM + LSCP4	LSTM + Isolation Forest	LSTM + OCSVM
Experiment 2.1	Confusion Matrix : [[4537 45] [2 25]]	Confusion Matrix : [[4556 26] [2 25]]	Confusion Matrix : [[4560 22] [2 25]]	Confusion Matrix : [[4559 23] [3 24]]	Confusion Matrix : [[4560 22] [2 25]]	Confusion Matrix : [[4569 13] [3 24]]	Confusion Matrix : [[4560 22] [2 25]]	Confusion Matrix : [[4549 33] [13 14]]

Table 7.11: Experiment- 2.1 Results: Confusion Matrix of all the tested approaches for anomaly detection on the test data

Experiment 2.1 Methods	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.357	0.926	0.990	0.515	0.990	45 + 2 = 47
LSTM + Threshold 10	0.490	0.926	0.994	0.641	0.994	26 + 2 = 28
LSTM + LSCP1	0.532	0.926	0.995	0.676	0.995	22 + 2 = 24
LSTM + LSCP2	0.511	0.889	0.995	0.649	0.994	23 + 3 = 26
LSTM + LSCP3	0.532	0.926	0.995	0.676	0.995	22 + 2 = 24
LSTM + LSCP4	0.649	0.889	0.997	0.750	0.997	13 + 3 = 16
LSTM + Isolation Forest	0.532	0.926	0.995	0.676	0.995	22 + 2 = 24
LSTM + One-Class SVM	0.298	0.519	0.993	0.378	0.990	33 + 13 = 46

Table 7.12: Experiment - 2.1 Results: Using Evaluation Metrics

In experiment 2.1, 'LSTM + LSCP-4' has achieved a higher F1-Score of 75% followed by LSTM + Isolation Forest' , 'LSTM + LSCP-1' and 'LSTM + LSCP-3' all with an F1-Score of 67.6% respectively. F1-Score of 'LSTM + Threshold10' is 64.1% which is quite low compared to the 'LSTM + LSCP-4' as given in Table 7.12. 'LSTM + OC-SCM' and 'LSTM + Threshold-5' yet again performed poorly with lowest F1-Scores of 37.8% and 51.5% respectively.

7.3.2.2 Experiment 2.2 - Multi-Step

For the multi-step multivariate LSTM model, a prediction length of 12 is used for a lookback of 12 steps. The LSTM model had 5 recurrent layers in total with a decreasing number of neurons [512, 256, 128, 64, 32], a dropout of 20%, and trained using Adam optimizer with 0.0001 learning rate, and a batch size of 1024 for 300 epochs, as given in Table 7.13. The forecast errors were further fed to the mentioned unsupervised outlier detection approaches.

Experiments	Model Architecture	Optimizer	Epochs	Batch Size	Lookback	Future Steps	RMSE
Experiment 2.2	LSTM (512) Dropout (0.2) LSTM (256) Dropout (0.2) LSTM (128) Dropout (0.2) LSTM (64) Dropout (0.2) LSTM (32) Dense	Adam: learning rate=0.0001 beta_1=0.9 beta_2=0.999 epsilon=1e-08 decay=0.0	300	1024	12	12	Train Score: 101.42 RMSE Test Score: 141.08 RMSE

Table 7.13: LSTM Model details for Muti-Step Multivariate data.

For experiment 2.2, all the unsupervised techniques with outlier fraction 0.01 were used. The configurations of LSCP-1, LSCP-2, and LSCP-3 were the same as the previous experiments (2.1, 1.2) except for LSCP-4. In LSCP-4 ensemble, outlier fraction was set to 0.009 and five LOFs each with neighbors 12, 24, 36, 60, 72 respectively, KNN with 48 ‘neighbors’ parameter and Isolation forest with 0.01 contamination were used in Experiment 2.2.

Experiments	LSTM + Threshold 5	LSTM + Threshold 10	LSTM + LSCP1	LSTM + LSCP2	LSTM + LSCP3	LSTM + LSCP4	LSTM + Isolation Forest	LSTM + OCSVM
Experiment 2.2	Confusion Matrix : [[4526 56] [8 19]]	Confusion Matrix : [[4549 33] [10 17]]	Confusion Matrix : [[4543 39] [19 8]]	Confusion Matrix : [[4545 37] [17 10]]	Confusion Matrix : [[4548 34] [14 13]]	Confusion Matrix : [[4560 22] [12 15]]	Confusion Matrix : [[4552 30] [10 17]]	Confusion Matrix : [[4538 44] [22 5]]

Table 7.14: Experiment - 2.2 Results: Confusion Matrix of all the tested approaches for anomaly detection on the test data

Experiment 2.2 Methods	Precision	Recall	Specificity	F1-Score	Accuracy	Number of Misclassified Points (FP + FN)
LSTM + Threshold 5	0.253	0.704	0.988	0.373	0.986	56 + 8 = 64
LSTM + Threshold 10	0.340	0.630	0.993	0.442	0.991	33 + 10 = 43
LSTM + LSCP1	0.170	0.296	0.991	0.216	0.987	39 + 19 = 58
LSTM + LSCP2	0.213	0.370	0.992	0.270	0.988	37 + 17 = 54
LSTM + LSCP3	0.277	0.481	0.993	0.351	0.990	34 + 14 = 48
LSTM + LSCP4	0.405	0.556	0.995	0.469	0.993	22 + 12 = 34
LSTM + Isolation Forest	0.362	0.630	0.993	0.459	0.991	30 + 10 = 40
LSTM + One-Class SVM	0.102	0.185	0.990	0.132	0.986	44 + 22 = 66

Table 7.15: Experiment- 2.2 Results: Using Evaluation Metrics

As per results for experiment 2.2 in Table 7.15, all the models have performed poorly, indicating that multistep prediction for multivariate data is not suitable for the detection process. The reason could be an increase in the complexity of the model due to the sequencing of multiple features for multiple time steps. On comparing the F1-Scores of models for this approach, the ranking for the F1-Scores of all the approach goes as follows: 'LSTM + LSCP-4' > 'LSTM + Isolation Forest' > 'LSTM + Threshold10' > 'LSTM + Threshold5' > 'LSTM + LSCP-3' > 'LSTM + LSCP-2' > 'LSTM + LSCP-1' > 'LSTM + OC-SCM'.

7.4 Experiments on the Localization Module

As discussed in Chapter 6, the localization module consists of a mapper and three detectors for each of the GraphQL's spans or sub-components - POST, GET and Cache respectively to detect performance anomalies for these spans. The spans corresponding to GraphQL data are extracted, pre-processed, and fed to each of its respective detectors as discussed in the previous chapter. The mapper receives the anomaly results from the Detection Module (for GraphQL), the three individual detectors (for each of GraphQL sub-components: POST, GET, Cache), and maps the timestamps of GraphQL anomalies with sub-component anomalies to identify the sub-component that is responsible for degrading GraphQL service's performance.

For this experiment, we considered the data from 18th June to 13th August 2019 to perform both the GraphQL anomaly detection (Section 7.3) by feeding the GraphQL data to the Detector module and perform localization by feeding GraphQL spans' data to the Localizer module. As per the experiments on the Detector module (Section 7.3), we chose the 'LSTM + LSCP-4' ensemble approach to detect the performance anomalies of GraphQL data (average response time of GraphQL) as shown in Figure 7.11. Multivariate data is used for building the Single-Step model as explained in experiment 2.1. The data was split into training and testing sets, where data from 18th June - 15th July 2019 was used for training and remaining data for testing the model. 20% of the training data was used as a validation set.

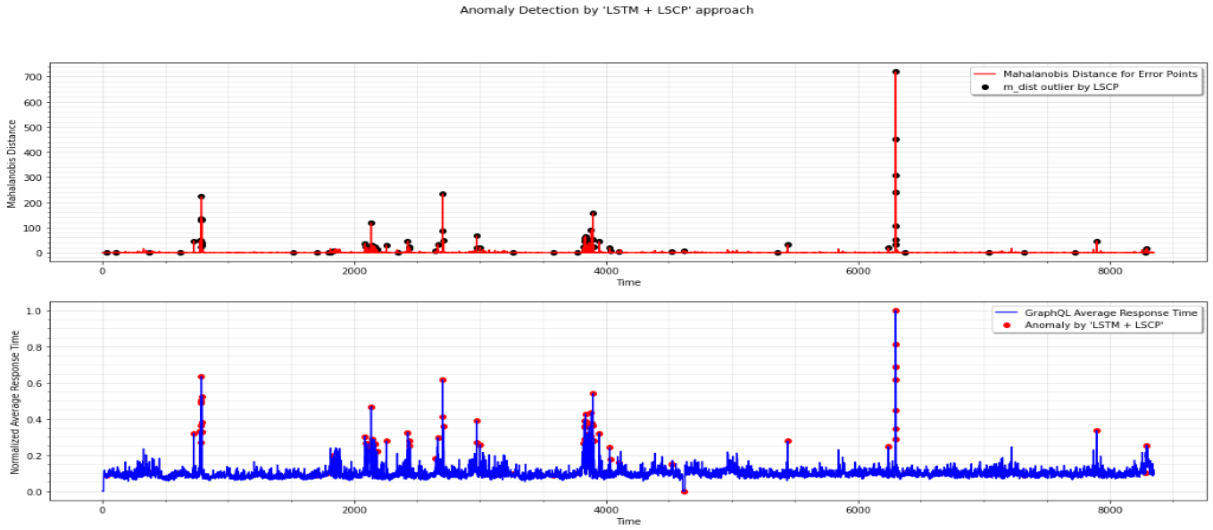


Figure 7.11: Anomalies for GraphQL's average response time data detected by Detector Module

Simultaneously, the spans of GraphQL data are fed to their respective individual detectors to detect performance anomalies for sub-components (POST average response time, GET average response time, Cache average response time). The three individual detectors also use the same 'LSTM + LSCP' approach, but with different configurations to forecast the expected average response time and detect anomalies when the actual value deviates from the expected baseline for each of the sub-components respectively using the unsupervised outlier detection method. The anomalies detected for each of the sub-components are shown in Figures 7.12, Figure 7.13, and Figure 7.14.

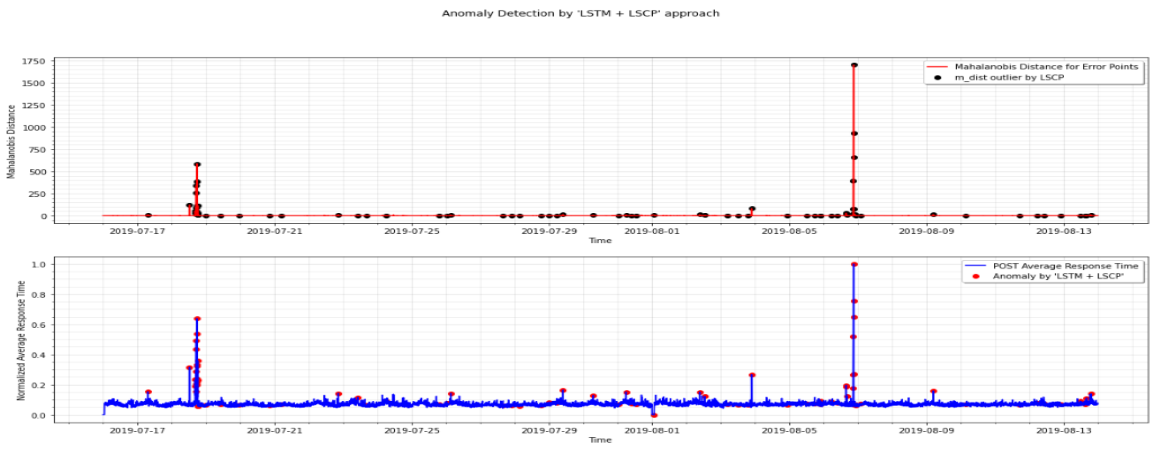


Figure 7.12: Anomalies for POST average response time data detected by detector 1

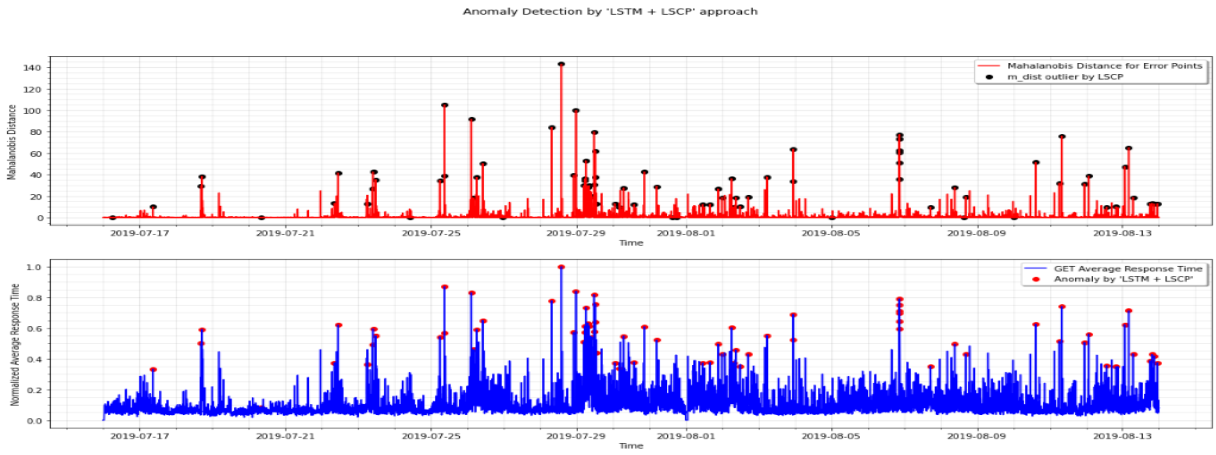


Figure 7.13: Anomalies for GET average response time data detected by detector 2

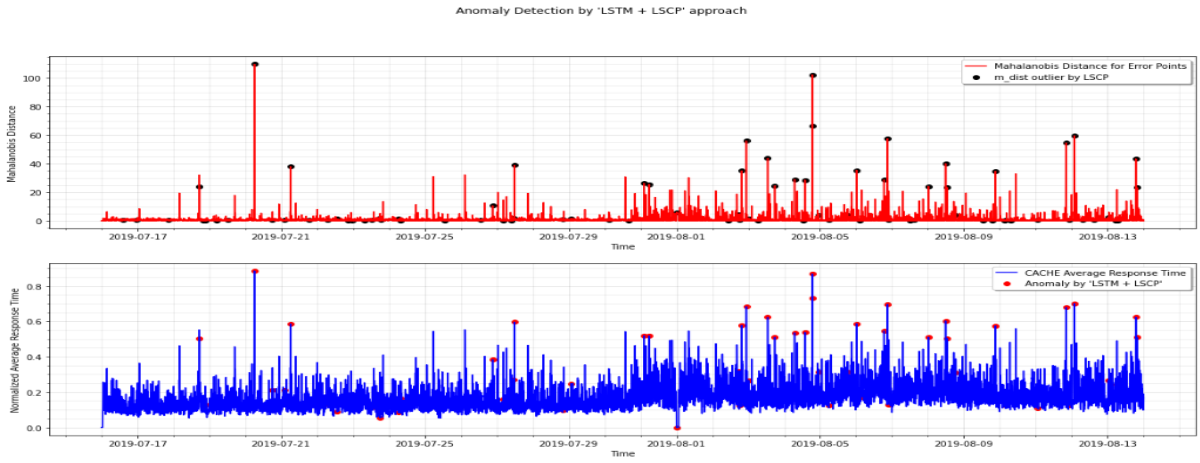


Figure 7.14: Anomalies for CACHE average response time data detected by detector 3

The detected anomalies for GraphQL service from the Detector module and the detected anomalies for each of the 3 sub-components from detectors in localizer module is fed to a mapper function which uses a simple logic of mapping the timestamps of GraphQL anomalous points with the timestamps of each of the anomalous points of POST, GET and Cache, to identify which of the sub-components is responsible for degrading the performance of the GraphQL service by comparing the time of occurrence of anomalies. The identified causal components by the mapper are later sent as a report to the alert management system for further action. The report includes the list of anomalous intervals of GraphQL service along with the details of root-cause which are described using a directed graph or DiGraph representations using the networkx library in python as displayed below in Figure 7.15.

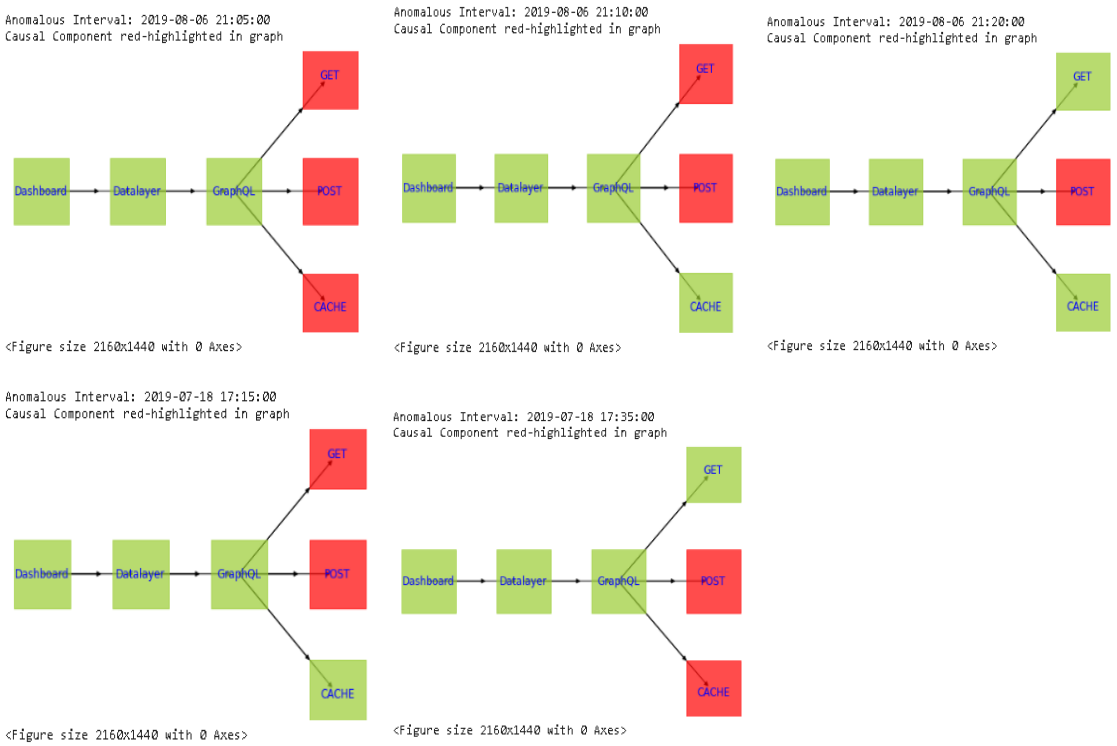


Figure 7.15: Mapper generated report which consists of details of GraphQL anomalous interval, i.e., the time interval at which anomaly occurred along with DiGraphs representation by highlighting the root-cause or causal components in red color.

As shown in Figure 7.15, the causal components for GraphQL performance anomalies are highlighted in red. There could be a single causal component or more than one component

responsible for GraphQL performance issues as shown in the results above. Based on our experiments, we observed a few results from the mapper, where the reason or root-cause for the GraphQL anomaly was not known, as depicted in Figure 7.16. There could be a few reasons such as:

i) Either the GraphQL anomaly detected by the Detector Module might be a false anomaly (not a real anomaly) hence, none of the sub-components were highlighted as the root-cause.
ii) The GraphQL anomaly detected by the Detector Module could be a true anomaly, but the detector for sub-components might have failed to detect all of its true anomalies and hence, none of the sub-components were identified as a root-cause by the mapper, or
iii) Both the GraphQL anomaly from detector Module and anomalies detected for sub-components could be accurate, but the root-cause could be an external factor other than its sub-components (POST, GET, Cache).

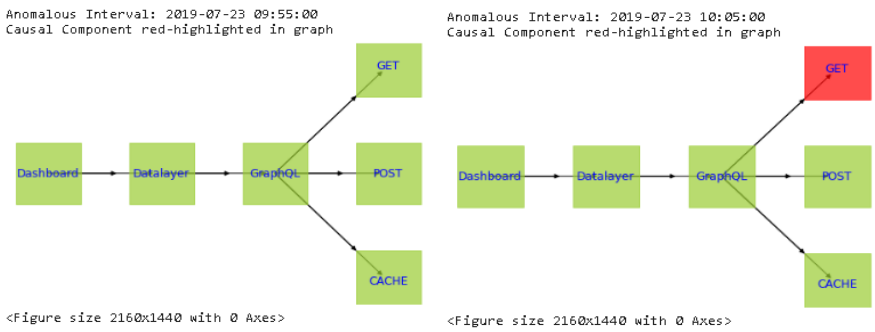


Figure 7.16: Mapper results where the root-cause of GraphQL anomaly is unknown (left).

7.5 Information Module

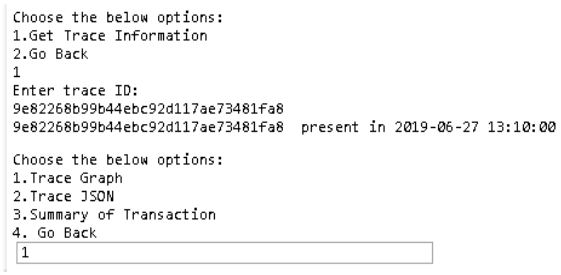
As discussed in Chapter 6, the Information module allows the user to access information about tracing events. This module provides 3 options:

i) Trace Graph: generates directed acyclic graphs (DAG) for a given transaction or trace ID along with the details of the parent-child relationship between the spans.

ii) Trace JSON: generates trace events in JSON format similar to how it is stored in the ES system.

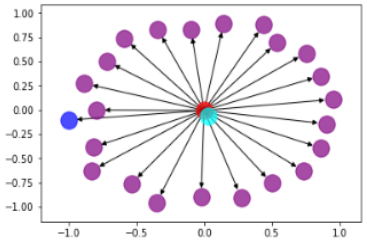
iii) Summary of Trace: provides detailed information of a transaction such as when the transaction happened, number of nodes or spans involved, their IDs and the time taken by each of spans, how nodes are interconnected by describing its parent-child relationship.

Figure 7.17 and Figure 7.18 displays the output of different options provided by the Information Module. In Figure 7.17, the directed acyclic graph includes nodes or spans starting from Datalayer(cyan colored node) followed by GraphQL(red node) with requests later propagating to POST (blue node), GET(green node), Cache (purple node) components. Figure 7.18 shows the output of the Trace JSON option and provides a summary of the transaction.

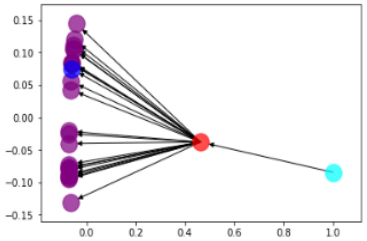


```
Request Flow Graph for the entered trace ID: 9e82268b99b44ebc92d117ae73481fa8


Parent-Child: {'791cdc24c19f2cc0': ['f9d56abceb719229'], 'f9d56abceb719229': ['488c81870be2e7e9', '5246be083e6c9d9b',
'707ccfe8a0e8424a', '683e6b751c7fdb9d', 'b180003a0a441e66', 'ef698353ded962fd', 'd72789ab7ec22017', '7b53f201d169358
e', 'd9394883a2ab7f0b', 'f8a62142bbb39441', '583c5c8bae09a53b', '74f60857c5ce0c35', 'eeebf82137356be4', 'bb1bbe4c9339
3e43', 'c5045634b747c44c', '965583a24999cf80', '38a6cbaacceae49f', '277ae1615fba55a9', '508c126f9b43fb79', '625a278f7
cd51de1', 'cd657474db82cf2f', 'eae107f278a148cd', '3d3d44651206ffd0']}
791cdc24c19f2cc0
f9d56abceb719229
```



```
<Figure size 1800x1080 with 0 Axes>
```



```
<Figure size 1800x1080 with 0 Axes>

Choose the below options:
1.Trace Graph
2.Trace JSON
3.Summary of Transaction
4. Go Back
```

Figure 7.17: Trace Graph option generates DAG for the entered transaction ID.

```
Choose the below options:
1.Trace Graph
2.Trace JSON
3.Summary of Transaction
4. Go Back
3

**************************Summary of every single transaction****************************************

Transaction ID: 9e82268b99b44ebc92d117ae73481fa8 Number of Nodes in transaction: 25

Nodes and Time Taken by Nodes (in ms):
 {'GraphQL-f9d56abceb719229': 156.406, 'CACHE-488c81870be2e7e9': 1.688, 'CACHE-5246be083e6c9d9b': 1.178, 'CACHE-707cc
fe8a0e8424a': 1.263, 'CACHE-683e6b751c7fdb9d': 0.605, 'CACHE-b180003a0a441e66': 0.491, 'CACHE-ef698353ded962fd': 0.29
4, 'CACHE-d72789ab7ec22017': 1.367, 'CACHE-7b53f201d169358e': 1.026, 'POST-d9394883a2ab7f0b': 656.793, 'CACHE-f8a6214
2bbb39441': 2.74, 'CACHE-583c5c8bae09a53b': 2.003, 'CACHE-74f60857c5ce0c35': 0.878, 'CACHE-eeebf82137356be4': 0.651,
'CACHE-bb1bbe4c93393e43': 1.619, 'CACHE-c5045634b747c44c': 1.236, 'CACHE-965583a24999cf80': 1.133, 'CACHE-38a6cbaacce
ae49f': 0.423, 'CACHE-277ae1615fba55a9': 1.44, 'CACHE-508c126f9b43fb79': 0.774, 'CACHE-625a278f7cd51de1': 0.922, 'CAC
HE-cd657474db82cf2f': 0.826, 'CACHE-eae107f278a148cd': 3.619, 'CACHE-3d3d44651206ffd0': 3.525}

Parent-Child: {'791cdc24c19f2cc0': ['f9d56abceb719229'], 'f9d56abceb719229': ['488c81870be2e7e9', '5246be083e6c9d9b',
'707ccfe8a0e8424a', '683e6b751c7fdb9d', 'b180003a0a441e66', 'ef698353ded962fd', 'd72789ab7ec22017', '7b53f201d169358
e', 'd9394883a2ab7f0b', 'f8a62142bbb39441', '583c5c8bae09a53b', '74f60857c5ce0c35', 'eeebf82137356be4', 'bb1bbe4c9339
3e43', 'c5045634b747c44c', '965583a24999cf80', '38a6cbaacceae49f', '277ae1615fba55a9', '508c126f9b43fb79', '625a278f7
cd51de1', 'cd657474db82cf2f', 'eae107f278a148cd', '3d3d44651206ffd0']}

Choose the below options:
1.Trace Graph
2.Trace JSON
3.Summary of Transaction
4. Go Back
```

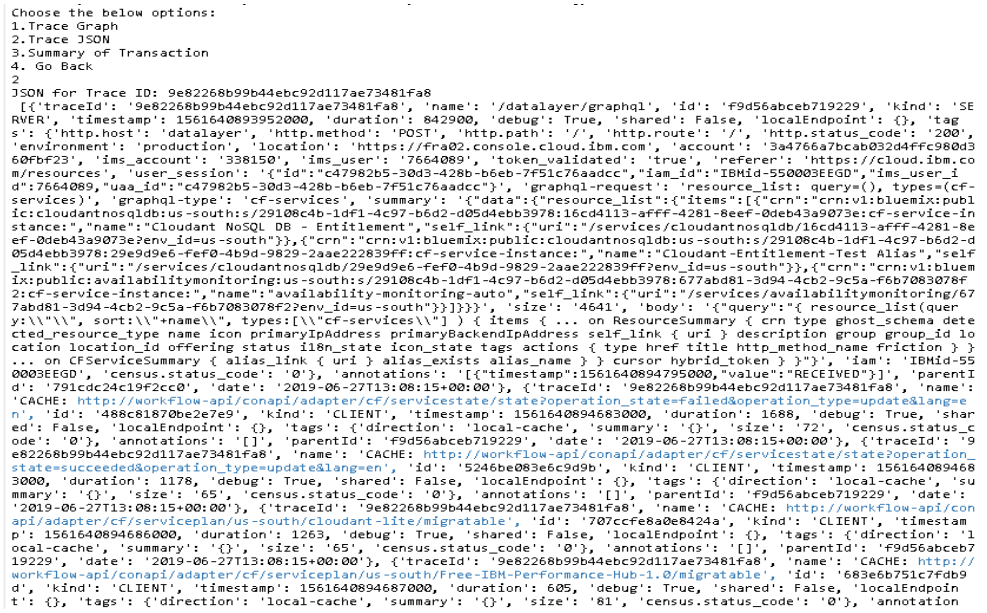


Figure 7.18: Output for Summary of Trace and Trace JSON options of Information Module.

7.6 Comparison with Related Work

The detection approach in our proposed system is a novel combination of LTSM and the unsupervised algorithms: (i) LSCP, (ii) Isolation Forest, and (iii) OC-SVM. The output of LSTM, i.e., forecast errors, was fed to the three unsupervised outlier detection algorithms to detect the outlying distance of error points from its distribution to further detect the anomalous data points, rather than explicitly setting a threshold on the '*m-dist'* distance value. Many existing approaches (identified in Table 7.16) have used such static thresholds irrespective of any algorithms (statistical/ supervised/unsupervised) they choose as discussed in Chapter 3 where the details of the existing approaches and the gaps are identified.

The static thresholding approach requires timely updates to its cut-off value as the input data changes over time. This might be even more tedious and time-consuming if there are various detection modules for every service. If there are hundreds and thousands of services, then keeping track of static threshold or cut-off values of detection modules for each service will make the monitoring system inefficient. In contrast, by using our

approach of the combination of LSTM and (i) LSCP, (ii) Isolation forest, and (iii) OC-SVM, there is no need to explicitly set the threshold on *m_dist* value instead it dynamically identifies the outlying *m_dist* values in order to identify its corresponding anomaly data points thus making the system entirely automated.

Author	**Proposed Model/Framework Name**	**Methods in Algorithms**	**Type of Learning**	**Data Used**
Vallis et. Al [48]	AdVec Algorithm	EDS + Statictical method + Piecewise approximation	Unsupervised	System Metrics and Application Metrics
Imam et. al [44]	Microsoft ML time series algorithm	ARTXP algorithm + ARIMA algorithm + Anomaly Index threshold	Unsupervised	Application Log Files
Sasho et al. [47]	-	Variational Autoencoders + Probability-based dynamic error thresholding	Unsupervised	Distributed traces
Haowen et al. [50]	Donut	Dimensionality reduction + Variational Auto-Encoder (VAE) + Threshold	Unsupervised	Application Metrics
Malhotra et al. [49]	-	LSTM + Likelihood estimation	Unsupervised	ECG, Space shuttle, power demand, and multi-sensor engine dataset.
Chen et al. [51]	SeqVL	Sequential VAE (+ Threshold) + LSTM	Unsupervised	KPI dataset and Yahoo dataset
Leandro et al. [52]	MULDER	Surprise Metric + (10th & 90th) Percentiles + 3-Standard deviation test	Unsupervised	NAB dataset (univariate)
Bikash et al. [36]	Cloud PD	HMM + Correlation Analysis, KNN	Statistical Method	System and application metrics
Daniel et. al [43]	-	SVM + Moving Average	Supervised	AWS dataset
Tian et al. [53], Tao et al. [54]	-	LOF	Unsupervised	Cloud applications [53], Workload patterns [54]
Sauvanaud et. al [46]	ADS (Anomaly Detection System)	Random Forests, Neural Networks, Nearest Neighbors, and Naive Bayes	Supervised	CPU, Memory, Disk, and Network performance data
Samir et al., in [60]	DLA (Detection and Localization System for Anomalies)	Hierarchical Hidden Markov Models (HHMM) + Correlation Analysis	Statistical Method	Performance data of (services, containers, nodes 'VM')

Qingfeng et al. [45]	ADS (Anomaly Detection System)	DTW + (SVM, Naïve Bayes, Nearest Neighbors, and Random forests)algorithm	Supervised	Performance data of microservice containers
Mohsen et al [56]	DeepAnt	CNN + Euclidean Distance	Unsupervised	Yahoo dataset, NAB dataset
Gu et al.[32] ,Tan et al.[31]	-	Markov Chain model + Bayesian Classifier	Statistical Method	
James et. al in [58], and Kanishka et. Al in [59]. Breunig et al., in [55]	-	KNN	Unsupervised	
Xiao et al., in [61]	TaskInsight	Clustering	Unsupervised	System-level metrics, such as CPU and memory utilization

Table 7.16: List of existing approaches discussed in Chapter 3

A comparison of the performance of the three proposed novel combination approaches and existing static threshold approach was made for every experiment as described in Section 7.3 (see Table **7.4**, Table **7.7**, Table **7.8**, Table **7.9**, Table **7.12**, Table **7.15**). A summary of three topmost approaches has been tabulated as given below in Table 7.17. The results show that the proposed novel 'LSTM + LSCP-4' hybrid variant detector performs detection better with a higher F1-score than the existing static error threshold technique and other ensemble approaches across all the experiments.

Data	**Prediction Steps**	**Experiments**	**LSTM + LSCP4 F1-Score**	**LSTM + Isolation Forest F1-Score**	**LSTM + Threshold-10 F1-Score**
Univariate Data	**Single Step**	**Experiment 1.1**	91.70%	91.10%	92.10%
	Multi Step	**Experiment 1.2.1**	94.70%	90.00%	92.10%
		Experiment 1.2.2	76.30%	75.00%	71.10%
		Experiment 1.2.3	82.50%	80.00%	81.00%
Multivariate Data	**Single Step**	**Experiment 2.1**	75.00%	67.60%	64.10%
	Multi Step	**Experiment 2.2**	46.90%	45.90%	44.20%

Table 7.17: Summary of results across all experiments for the top 3 ensemble methods

As shown in Table 7.17, for the Univariate Single-Step experiment, the 'LSTM-LSCP-4' hybrid achieved an F1-Score of 91.7% and for Multi-Step experiments of 12, 24, 48 prediction time steps, the ensemble model performed detection with an F1-Score outcome

of 94.7%, 76%, 82.5% respectively. For Multivariate Single-Step, the 'LSTM-LSCP4' model achieved a moderate F1-Score of 75% whereas, for Multi-Step experiment of 12 prediction length, it scored comparatively less with an F1-Score of 46.9%, which was still higher than the F1-Scores of other tested ensembles. The ensemble methods: 'LSTM + Isolation Forest' and 'LSTM-Threshold10' performed moderately with a decent score of F1-Score metric whereas 'LSTM + OC-SVM' hybrid performed consistently bad resulting in low F1-Score measures.

The average scores of precision, recall, and F1-score measures were calculated for the top three models that we used across all the experiments under univariate data as described in Section 7.3.1 (see Experiment 1.1, 1.2.1, 1.2.2, 1.2.3).

Method	F1-Score Average	Precision Average	Recall Average
LSTM+Threshold 10	84.08%	77.17%	93.06%
LSTM+LSCP4	86.30%	82.29%	90.97%
LSTM+Isolation Forest	84.03%	76.61%	93.06%

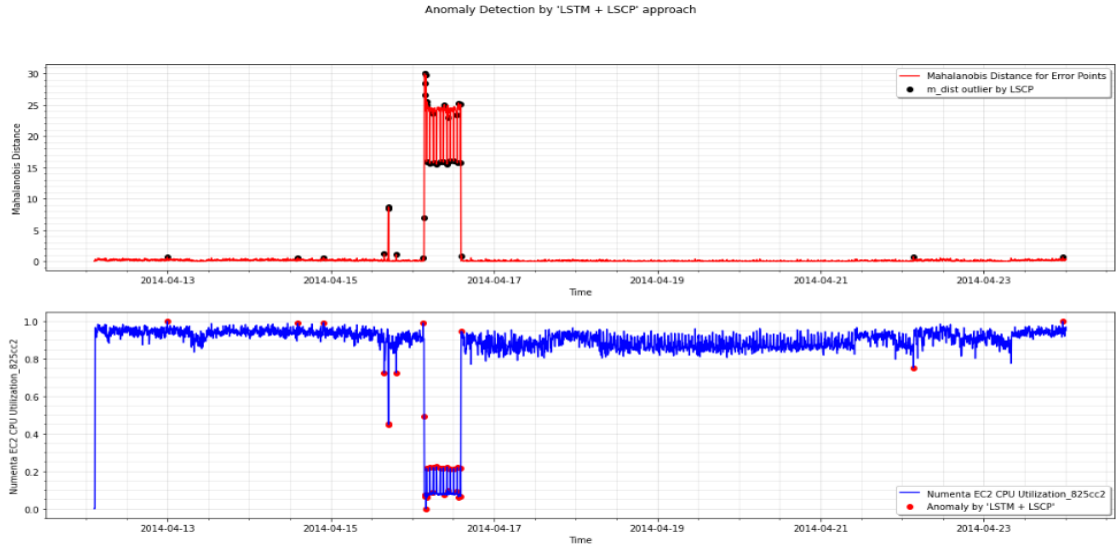
Table 7.18: Average of the metric scores across all the Univariate experiments

The average F1-Score of the 'LSTM+LSCP4' ensemble approach is the highest compared to the existing static thresholding – 'LSTM+Threshold-10' and the 'LSTM+Isolation' ensemble methods as shown in Table 7.18.

7.6.1 Experiment on Numenta Anomaly Benchmark (NAB) dataset 1

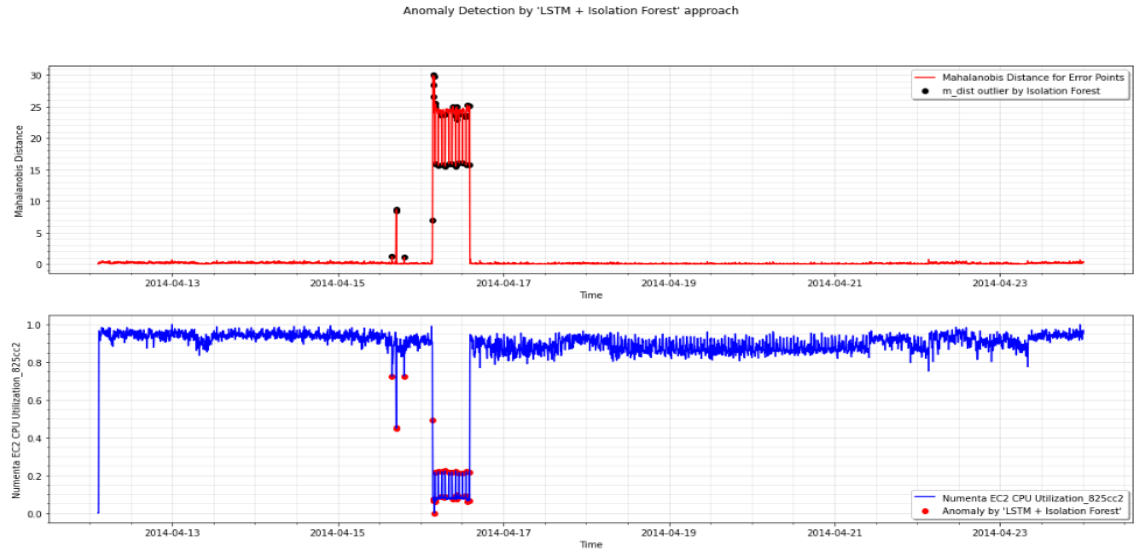
Since the above experiments were conducted on the third-party cloud system's dataset, we also evaluated the performance of our approach by experimenting on the Numenta Anomaly Benchmark (NAB) Dataset. NAB [108] platform provides access to real-world, labeled data files across multiple domains and is used by various researchers to validate the performance of their detection models. We used Numenta's "ec2_cpu_utilization_825cc2" dataset on our proposed approach to detect the anomalies and compared them with Numenta's results. The detection results of the proposed 'LSTM + LSCP-4' and 'LSTM +

Isolation Forest' hybrid methods results are shown in Figure 7.19 and Figure 7.20 respectively.




```
Confusion Matrix :
[[3073   12]
 [ 320   23]]
Accuracy Score : 0.9031505250875146
```

Figure 7.19: Detection results for 'LSTM + LSCP-4' hybrid method on NAB 'ec2 CPU utilization' dataset.




```
Confusion Matrix :
[[3076    9]
 [ 317   26]]
Accuracy Score : 0.9049008168028004
```

Figure 7.20: Detection results for 'LSTM + Isolation Forest' on NAB 'ec2 CPU utilization' dataset.

Based on Numenta's standard scores [109], Numenta's HTM model results in the following TP, TN, FN, and FP as shown in Figure 7.21.

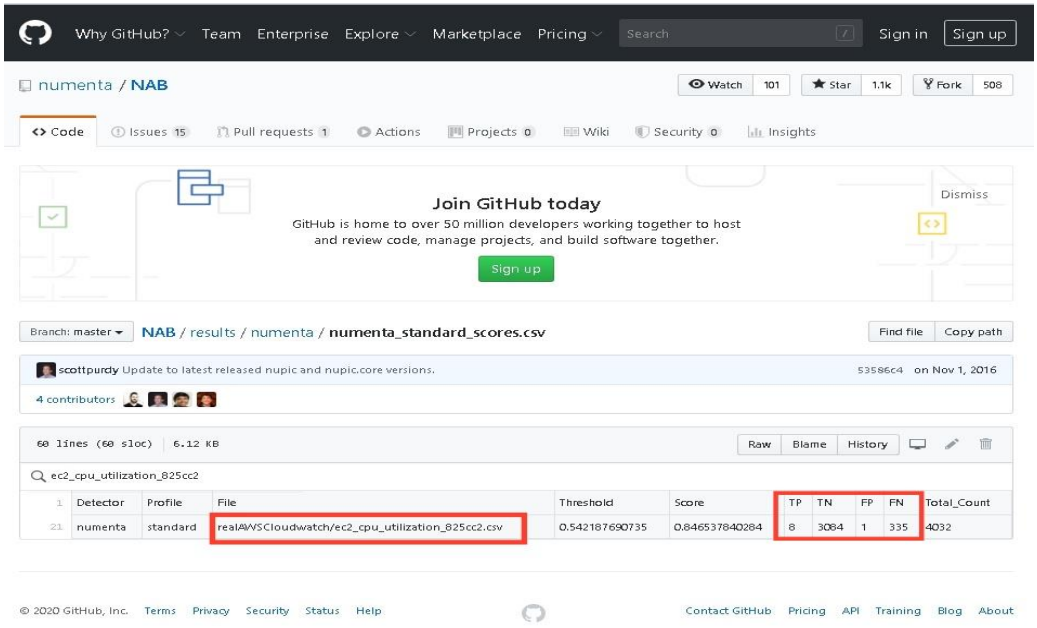


Figure 7.21: Detection results for Numenta's HTM model on NAB 'ec2 CPU utilization' dataset.

Method	Precision	Recall	Specificity	F1-Score	Accuracy
LSTM + Isolation Forest	0.743	0.076	0.997	0.138	0.905
LSTM + LSCP4	0.657	0.067	0.996	0.122	0.903
Numenta's HTM model	0.889	0.023	0.999	0.045	0.902

Table 7.19: Results for NAB 'ec2 CPU utilization' dataset.

On comparing, the results for the proposed combinations and Numenta's HTM benchmark scores as given in Table 7.19, we observe that the proposed novel combination for anomaly detection performs better than the Numenta's standard scores, where 'LSTM + Isolation forest' has higher F1-score of 13.8% followed by 'LSTM + LSCP-4' with F1-Score of 12.2%, whereas Numenta's HTM model results in a low F1-score of 4.5%.

7.6.2 Experiment on Numenta Anomaly Benchmark (NAB) dataset 2

We conducted another experiment on a different dataset from the NAB platform – "elb_request_count_8c0756" dataset. The detection results of the proposed 'LSTM + LSCP-4' and 'LSTM + Isolation Forest' hybrid methods results are shown in Figure 7.22 and Figure 7.23 respectively.

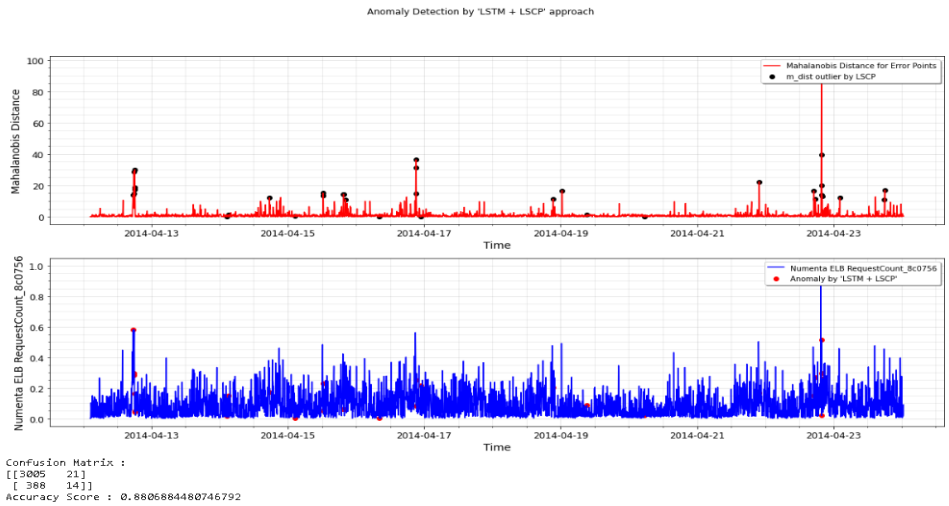


Figure 7.22: Detection results for 'LSTM + LSCP-4' hybrid method on NAB 'elb_request_count_8c0756' dataset

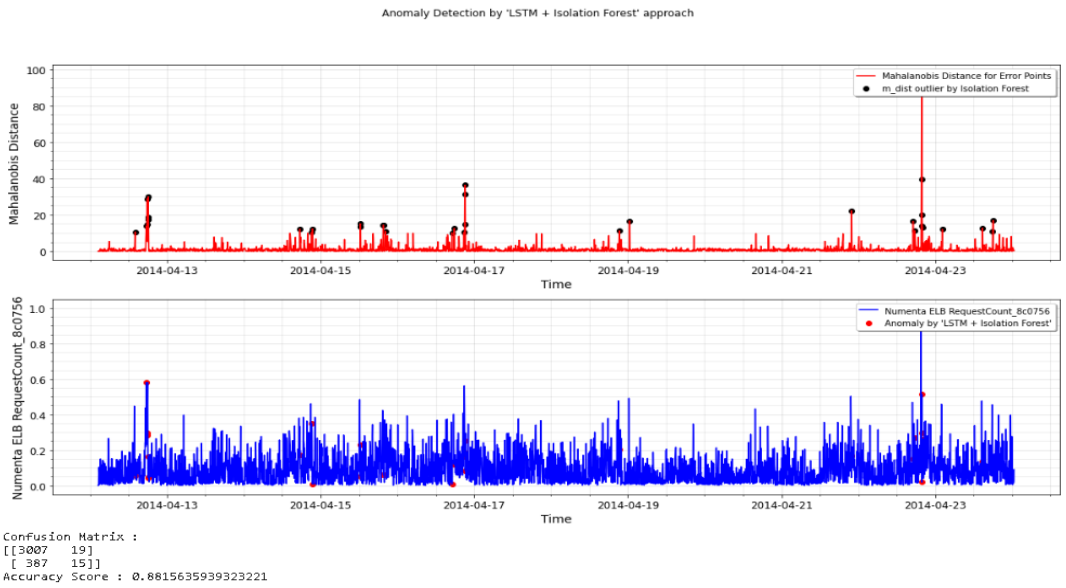


Figure 7.23: Detection results for 'LSTM + Isolation forest' hybrid method on NAB 'elb_request_count_8c0756' dataset.

Based on Numenta's standard scores [109], Numenta's HTM model results in the following TP, TN, FN, and FP as shown in Figure 7.24.

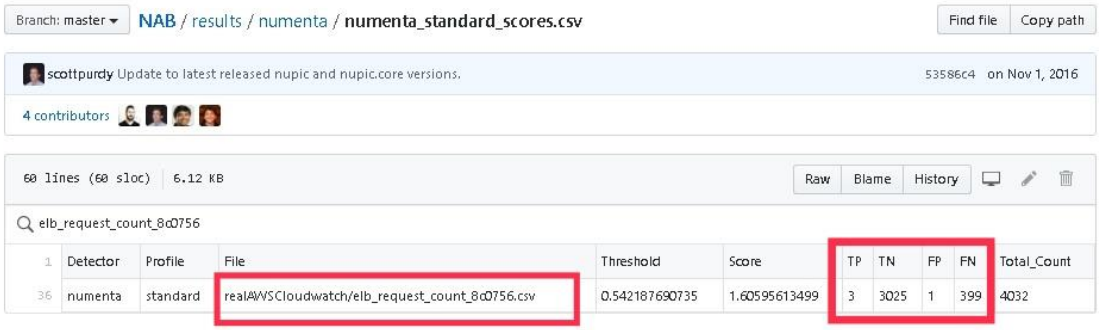


Figure 7.24: Detection results for Numenta's HTM model on NAB 'elb_request_count_8c0756' dataset.

Method	Precision	Recall	Specificity	F1-Score	Accuracy
LSTM + Isolation Forest	0.441	0.037	0.994	0.069	0.882
LSTM + LSCP4	0.5	0.051	0.993	0.093	0.881
Numenta's HTM model	0.75	0.007	1	0.015	0.883

Table 7.20: Results for NAB 'elb_request_count_8c0756' dataset

On comparing, the results for the proposed combinations and Numenta's HTM benchmark scores as given in Table 7.20, we observe that the proposed novel combination for anomaly detection performs better than the Numenta's standard scores, where 'LSTM + LSCP' has higher F1-score of 9.3% followed by 'LSTM + Isolation forest' with F1-Score of 6.9%, whereas Numenta's HTM model results in a low F1-score of 1.5%.

The experiments on both the NAB datasets show that our approaches resulted in low F1-scores, but it does perform better than the standard benchmark score of the Numenta HTM model.

7.7 System development and Testing

In this section, we describe the approach towards the development and testing of our proposed system. All the modules of the proposed system were developed individually as separate python programs. The development and testing for an individual module were performed simultaneously. For example, in the data extraction process (refer Section 6.2.2) the 5 steps were developed as a separate program where each script (say Step 2) was executed after the successful execution of another (Step 1). Unit testing was carried out for each program (steps 1-5) in the data extraction module, followed by Integration testing of all the programs (steps 1-5) within the extraction module. Similarly, the unit testing and integration testing was carried out for all the modules.

In the post-development phase, we conducted the verification and validation procedure for each of the scripts. Verification is an internal process to evaluate whether or not a product, service, or system complies with a requirement or specifications. Validation is an assurance that a product, service, or system meets the user's expectations.

Data Extraction Phase Verification: The TraceID collection and Get Trace Information step (see Section 6.2.2) was verified by comparing the number of transactions or trace IDs fetched by our program script with the total number of trace IDs that were shown in the ElasticSearch system of the third-party cloud system. Summary information (Total requests, Total Duration) was verified by comparing the graphical representations of the

features collected by our programs with the graphical representation for that dataset on the Kibana tool (visualization tool for ES).

Data Pre-processing Phase Verification: Post feature scaling the dataset was plotted and compared with the data in third-party through visualization plots in Kibana for the verification process.

The Detection process was verified using the visual plots and evaluation metrics such as F1-Score by making use of the label information. The detection results were validated by domain experts visually. The proposed novel approach for anomaly detection was also evaluated on two different benchmark datasets available on the NAB platform as shown in Sections 7.6.1 and 7.6.2.

7.8 Summary

In this chapter, we discuss tools and libraries used in the experiments, its implementation details, and their results for the proposed prediction-based automated hybrid anomaly detection and localization system.

In the experiments described in Section 7.3, the detector module of the system was implemented using the proposed novel hybrid approach, and the existing static error thresholding technique to compare the effectiveness of the proposed detection methodology. For time-series prediction, both univariate and multivariate data were used for the experiments. For both the variations of data, two different versions of the time-series LSTM models: Single-step and Multi-step LSTM, were built to perform prediction-based anomaly detection and then combined with the three unsupervised outlier detection algorithms to detect the outlying distance of error points from its distribution to further detect the anomalous data points. A total of 8 different hybrid models were tested: four LSTM with LSCP ensemble models, one LSTM with Isolation Forest, LSTM with One-Class SVM, and two LSTM with Static thresholding methods.

The implementation details of the localization module were also described and demonstrated to show how the root-cause of detected anomalies were identified for the

given dataset (see Section 7.4), and further, the functionality of the Information module was also described in Section 7.5.

A comparison of the proposed detection approach with the related study is described in Section 7.6, where the proposed novel detection approach is also tested on a benchmark dataset (NAB) and its results are compared with existing Numenta's HTM results, followed by the next section describing the System development and testing carried out in our work.

Chapter 8

8 Discussion, Conclusion and Future Work

In this chapter, we discuss the challenges, impact of our work, and different threats to validity. We conclude the thesis and describe future work that we plan to explore.

8.1 Discussion

In this section, we discuss the results and impact of our proposed system, the challenges that we encountered in our work, and the different aspects that we could have explored.

8.1.1 Impact of our proposed system

The proposed automated anomaly detection and localization system: (i) detects performance anomalies of a microservice through monitoring the performance metric data extracted from the tracing events using a novel approach of a prediction-based anomaly detection technique which combines a time-series model and unsupervised learning algorithms - LSCP, Isolation forest, and One-Class SVM, and (ii) locates the causal components for the detected anomalies. By using the proposed combination of LSTM and an unsupervised learning algorithm, there is no need to explicitly set the threshold on *m_dist* value. Instead, it dynamically identifies the outlying *m_dist* values in order to identify its corresponding anomaly data points by using an unsupervised outlier detection technique such as LSCP or Isolation Forest, thus making the system entirely automated, unlike the static rule-based thresholding approaches (see Table 7.16) which can be inefficient when the data or load varies over time, as discussed in Chapter 3 and Section 4.3. The proposed detection ensemble identifies point anomalies and also collective anomalies since we group the data into 5-minute intervals.

The output of the system is an anomaly report that consists of the time-interval of the occurrence of the anomaly and a directed acyclic graph highlighting its causal component, as shown in Figure 7.15.

The proposed system is not limited to GraphQL service as shown in the demonstration of our work but can be adapted to other services as well for time-series anomaly detection.

In a microservices-based cloud environment where multiple instances of all the microservices need to be monitored at once, which requires to check the logs of multiple services and track one user request through multiple systems, the proposed system is quite capable to handle such cases while easing the monitoring process compared to the existing framework (see Section 4.2) which might not be as reliable or simple.

In a microservices-based cloud system with numerous microservices and its multiple instances, the proposed system helps to in locating the root cause of an anomaly and overcomes the challenges of SREs in manually analyzing and tracking the root-cause or faulty microservice/component.

8.1.2 Challenges

The biggest challenge that we faced during our work was the absence of label information. The availability of labeled data and benchmark datasets is a major issue in the field of anomaly detection research in general. Many research works [45, 46] have dealt with this issue by training a model on normal data and then injecting artificial anomalies in the dataset and further verify if the trained model detects the injected anomalies or not. The problem with this approach is that anomalies are uncertain and might have different patterns due to which the trained model might not be effective for a real-world dataset. In our case, we used the real-world production data from a third-party cloud microservices, but the problem was the absence of label information.

Based on our exploratory data analysis, we observed the presence of anomalies in our data which is quite obvious, given the fact that the data was extracted from a production environment. As a result, it was challenging to train a model using production data as there were chances that the model would learn the anomaly patterns along with the normal data and later identify an anomaly as a normal data point. To mitigate this effect, we collected data for a long duration (six months) in order to identify the portion of data which seemed

normal and sufficient enough for training a deep learning model, under the guidance of domain experts.

8.1.3 Alternate analysis/aspects for consideration

As far as the localization technique is considered, the list of possible root-causes is limited only to its calls to sub-components or child nodes. For example, for GraphQL service the localization module limits the causal components to POST, GET, and Cache, but there might be other external factors that could be a reason for degrading GraphQL service's performance such as an outage for a certain interval of time might result in lower peaks or even '0' requests, causing the average response time of GraphQL to drop to '0' during that interval. With no further requests propagated to its sub-components, this might show the response times for them (POST, GET, and Cache) to be '0'. In such cases, as per the logic used by the mapper, the causal component for GraphQL's performance degradation issue will either be POST/GET/Cache or all of them respectively, but the actual root-cause here tends to be an overall cloud outage. Similarly, there could be other external factors that are responsible for GraphQL downtime apart from its dependency on sub-components - POST/GET/Cache.

The proposed detection technique was tested on datasets that exhibited additive seasonality. However, an alternative could be to test the proposed hybrid model on datasets with multiplicative seasonality or different kind of trend patterns in a time-series, such as the exponential trend in the rise of COVID-19 cases.

Also, apart from creating a baseline for the 'average response time' feature, a baseline for the 'total number of requests' (request arrival pattern) and 'error rate' pattern for each of the response codes could be generated and used for analysis to verify how such patterns could help improve the detection accuracy. The proposed detection ensemble identifies point anomalies and also collective anomalies since we group the data into 5-minute intervals. The detection technique can also consider other contextual or external factors to detect the contextual anomalies.

8.2 Threats to validity

1. The data used in our experiments are specific to GraphQL service and hence, the results will vary for other services' data. The data feature used for our experiments is Average response time and other time-related features in case of multivariate data and hence the model should be reconfigured and tuned for different services and other metrics such as CPU, memory usage, as input features.
2. The data used for our experiment lies between June to December 2019. There was a notable change in the data from June to September 2019 due to various factors like production deployment, system configurations, etc., during that period. Hence, the model was configured for the last portion of the data. It is assumed that there will not be a huge notable difference in the average response time feature of the data based on our testing (Dual Validation in Section 7.3.1.1).
3. The model seems capable and promising for real-time anomaly detection, but it should be deployed in the production environment and tested for real-time streaming data for better assessment. The speed of the data extraction as 5-minutes interval and pre-processing stages might be slow and result in delays during real-time detection and might require optimization to speed up the process.
4. The proposed work assumes that the causal components are directly dependent on the GraphQL service, which are basically its spans or sub-components. As discussed in the previous section, there could be external factors degrading the GraphQL service's performance.
5. The proposed detection method is capable of detecting point-based anomalies and collective anomalies, but not contextual anomalies.

8.3 Conclusion

In this thesis, we developed an automated prediction-based anomaly detection and localization system, (i) which detects performance anomalies using a time-series deep learning model and an ensemble of unsupervised learning techniques that can handle a huge volume of data generated from each of the individual microservices and avoid the burden of static thresholding approach that is used in the existing monitoring framework

and other literature works as discussed in Chapter 3 and 4, and (ii) identifies the casual components of the detected anomalies.

The motivation behind this research is to make the performance anomaly detection along with the localization process of the root causes smooth and accurate with no intervention required by the operations team. This idea is not a contribution to a specific service or product or cloud-based systems but to all fields from different domains where the data is time-series related data. Unlike existing techniques, our proposed system doesn't require any static thresholds or cut-offs for scoring the anomalies and is completely automated.

We developed the proposed system which consists of five modules: Data Extraction, Data Pre-processing, Anomaly Detection module, Localization module, and Information module. The novelty of the proposed system lies in the anomaly detection process of the Detection module that uses a novel combination of LSTM and unsupervised learning algorithms: (i) LSCP, (ii) Isolation forest, (iii) One-Class SVM, which avoids static thresholding to score the anomaly and follows a dynamic approach using the unsupervised learning algorithms. Another aspect of novelty in this work is using the LSCP model for univariate data, which hasn't been tested in the existing work previously.

We conducted experiments on a real-world time-series dataset of microservices from the production environment and the Numenta Anomaly Benchmark dataset. Based on our results on production dataset, we conclude that the proposed novel combination of 'LSTM + LSCP' and 'LSTM + Isolation Forest' are effective anomaly detectors with average F1-scores 86% and 84%, average precision scores of 82 % and 77%, and average recall scores of 91% and 93% respectively. They use the concept of prediction-based automated anomaly detection without having to explicitly set any thresholds or cut-offs for scoring anomalies and instead use unsupervised outlier detection ensemble – LSCP, Isolation forest for identifying anomalies. The localization module was also experimented using data from a third-party cloud system.

We wanted to experiment how the ensemble performs when LSTM includes additional features (day of the week, weekday/weekend, holiday/non-holiday, etc.,.) extracted out of timestamps as input even though LSTM is a time-series model in itself which learns

complex dynamics within the temporal ordering of input sequences. Based on our results, it shows that the ensemble experimented using multivariate data for both single-step and multi-step LSTM models did not perform as good as ensembles using univariate LSTM models. This concludes that the inclusion of timely features to an LSTM does not improve the performance of the LSTM model.

In our work, the production data used was specific to GraphQL service and its sub-components – POST, GET, Cache. However, our proposed system is not limited to GraphQL and its spans alone and can be used for any time-series data in a microservices-based cloud environment.

8.4 Future Work

Here, we list several future work items:

1. The main problem in the anomaly detection field is the absence of label information which creates a problem in determining the effectiveness of the approach. Labeling the data manually in order to evaluate the detection technique is a challenging task. Hence, to generate label information, a feedback loop can be implemented where a domain expert would send feedback regarding the detected anomalies if the detected anomalies are true or false. Based on domain experts' input the model can retrain using the feedback and learn the behavior of the data to predict anomalies during its encounter in the real-time production environment.
2. Further, the nature of data varies with time. Hence, the model requires an update or re-training. But doing this manually on a regular basis would be tedious and hence automatic re-training will be required which can be done when the label information is available.
3. Also, one can test the proposed system for real-time anomaly detection by implementing it in a production environment once the set-up and system integration are done. The proposed detection technique can be tested on different varieties of benchmark time-series datasets.
4. The data related to contextual or external factors responsible for anomalies in the localization module can be collected for future analysis and features such as error

rates, not considered in the current version of our proposed system, can be taken into account.

References

1. V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. ACM Computing Surveys.41(3), pp. 1-72, DOI: 10.1145/1541880.1541882, July 2009.
2. Tahir Mehmood, Helmi B Md Rais. Machine Learning Algorithms in Context of Intrusion Detection. 3rd International Conference on Computer and Information Sciences (ICCOINS), 2016.
3. Expert System team. "What is Machine Learning? A definition". https://expertsystem.com/machine-learning-definition/, March 2017. Accessed 2020-01-10.
4. Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. Data Mining: Practical machine learning tools and techniques. Morgan Kaufmann, 2016.
5. https://en.wikipedia.org/wiki/Semi-supervised_learning. Accessed: 2020-01-10.
6. Sciforce. "Anomaly Detection, A Key Task for AI and Machine Learning, Explained". https://www.kdnuggets.com/2019/10/anomaly-detection-explained.html, October 2019. Accessed 2020-01-10.
7. Louis Columbus, "State Of Enterprise Cloud Computing, 2018". https://www.forbes.com/sites/louiscolumbus/2018/08/30/state-of-enterprise-cloud-computing-2018/#5cde816265e3. Accessed: 2020-01-10.
8. Tom Huston, "Monitoring microservices". https://smartbear.com/learn/performance-monitoring/monitoring-microservices/. Accessed: 2020-01-16
9. Kamal Jacob, "The Significance of Microservices in Cloud Computing". https://www.manipalprolearn.com/blog/significance-microservices-cloud-computing/. Accessed: 2020-01-16.
10. Romana Gnatyk, "Microservices vs Monoliths: which architecture is the best choice for your business?" https://www.n-ix.com/microservices-vs-monolith-which-architecture-best-choice-your-business/, October 2018. Accessed: 2020-01-16.
11. Vinod KeshaoraoPachghare. Microservices Architecture for Cloud Computing. Journal of Information Technology and Sciences. Volume 2 Issue 1, 2016.
12. "The Future of Application Development and Delivery Is Now". https://www.nginx.com/resources/library/app-dev-survey/. Accessed: 2020-01-16.
13. Dave Swersky, "The Hows, Whys and Whats of Monitoring Microservices". https://thenewstack.io/the-hows-whys-and-whats-of-monitoring-microservices/, 2018. Accessed: 2020-01-16.
14. Apurva Dave and Loris Degioanni, "The five principles of monitoring microservices". https://thenewstack.io/five-principles-monitoring-microservices/, September 2016. Accessed: 2020-01-16.
15. H. Malik and E. M. Shakshuki, "Classification of Post-deployment Performance Diagnostic Techniques for Large-scale Software Systems," Procedia Computer Science, vol. 37, pp. 244–251, 2014.
16. Olumuyiwa Ibidunmoye, "Performance Anomaly Detection and Resolution for Autonomous Clouds", December 2017.

17. Kaya Ismail, "7 Tech Giants embracing microservices". https://www.cmswire.com/information-management/7-tech-giants-embracing-microservices/, August 2018. Accessed 2020-01-17.
18. Joseph Tsidulko, "The 10 Biggest Cloud Outages of 2019 (so far)". https://www.crn.com/slide-shows/cloud/the-10-biggest-cloud-outages-of-2019-so-far-/4, July 2019. Accessed: 2020-01-23.
19. Debjani Chaudhury, "Strange Cloud Outages that shook up the tech world in 2019". https://enterprisetalk.com/featured/strange-cloud-outages-that-shook-up-the-tech-world-in-2019/, December 2019. Accessed: 2020-01-23.
20. Syed Yousaf Shah, ZengwenYuany, Songwu Luy, Petros Zerfos. Dependency Analysis of Cloud Applications for Performance Monitoring using Recurrent Neural Networks. IEEE International Conference on Big Data (BIGDATA), 2017.
21. Beepa Bose, Joy Dutta, Subhasish Ghosh, Pradip Pramanick and Sarbani Roy. D&RSense : Detection of Driving Patterns and Road Anomalies. In 2018 3rd International Conference On Internet of Things: Smart Innovation and Usages (IoT-SIU). DOI: 10.1109/IoT-SIU.2018.8519861, February 2018.
22. Michael Walton, Maurice Ayache, Logan Straatemeier, Daniel Gebhardt, and Benjamin Migliori. Unsupervised Anomaly Detection for Digital Radio Frequency Transmissions. In 2017 16th IEEE International Conference on Machine Learning and Applications. DOI: 10.1109/ICMLA.2017.00-54, January 2018.
23. Haya Al-Thani, Hanadi Hassen, Somaya Al-maadeed, Noora Fetais, Ali Jaoua. Unsupervised Technique for Anomaly Detection in Qatar Stock Market. In 2018 International Conference on Computer and Applications (ICCA). DOI: 10.1109/COMAPP.2018.8460282, September 2018.
24. Vincent Vercruyssen,WannesMeert, Gust Verbruggen, Koen Maes, Ruben Bäumer, Jesse Davis. Semi-supervised Anomaly Detection with an Application to Water Analytics. In 2018 IEEE International Conference on Data Mining (ICDM). DOI: 10.1109/ICDM.2018.00068, December 2018.
25. Laurikkala, J., Juhola1, M., and Kentala., E. Informal identification of outliers in medical data. In Fifth International Workshop on Intelligent Data Analysis in Medicine and Pharmacology. 20-24, 2000.
26. Suzuki, E., Watanabe, T., Yokoi, H., and Takabayashi, K. Detecting interesting exceptions from medical test data with visual summarization. In Proceedings of the 3rd IEEE International Conference on Data Mining. 315-322, 2003.
27. Blender, R., Fraedrich, K., and Lunkeit, F. Identification of cyclone-track regimes in the North Atlantic. Quarterly Journal of the Royal Meteorological Society 123, 539, 727-741, 1997.
28. Escalante, H. J. A comparison of outlier detection algorithms for machine learning. In Proceedings of the International Conference on Communications in Computing. 2005.
29. O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance Anomaly Detection and Bottleneck Identification. ACM Computing Surveys, vol. 48, no. 1, pp. 4:1–4:35, July 2015.
30. Hiep Nguyen, Zhiming Shen, Yongmin Tan, and Xiaohui Gu. FChain: Toward black-box online fault localization for cloud systems. In Proceedings of the IEEE

33rd International Conference on Distributed Computing Systems (ICDCS'13). IEEE, 21–30. 2013.

31. Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems (ICDCS'12). IEEE, 285–294. 2012.
32. Xiaohui Gu and Haixun Wang. Online anomaly prediction for robust cluster systems. In Proceedings of the IEEE 25th International Conference on Data Engineering (ICDE'09). IEEE, 1000–1011. 2009.
33. Hui Kang, Xiaoyun Zhu, and Jennifer L. Wong. DAPA: diagnosing application performance anomalies for virtualized infrastructures. In Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services. 2012.
34. Lingyun Yang, Chuang Liu, Jennifer M. Schopf, and Ian Foster. Anomaly detection and diagnosis in grid environments. In Proceedings of the 2007 ACM/IEEE Conference on Supercomputing. IEEE, 1–9. 2007.
35. Joao Paulo Magalhaes and Luis Moura Silva. Detection of Performance Anomalies in Web-based Applications. In 9th IEEE International Symposium on Network Computing and Applications (NCA). IEEE, pp. 60–67. 2010.
36. Bikash Sharma, Praveen Jayachandran, Akshat Verma, and Chita R. Das. CloudPD: Problem determination and diagnosis in shared dynamic clouds. In Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13). IEEE, 1–12. 2013.
37. Manjula Peiris, James H. Hill, Jorgen Thelin, Sergey Bykov, Gabriel Kliot, and Christian Konig. PAD: Performance anomaly detection in multi-server distributed systems. In Proceedings of the 7th IEEE International Conference on Cloud Computing (CLOUD'14). IEEE. 2014.
38. Lizhe Wang, Samee U. Khan, Dan Chen, Joanna Kołodziej, Rajiv Ranjan, Cheng-zhong Xu, Albert Zomaya. Energy-aware parallel task scheduling in a cluster. Future Generation Computer Systems., vol. 29, no. 7, pp. 1661-1670. 2013.
39. R. Ahad, E. Chan, and A. Santos, "Toward Autonomic Cloud: Automatic Anomaly Detection and Resolution," in International Conference on Cloud and Autonomic Computing (ICCAC). IEEE, pp. 200– 203. 2015.
40. O. Ibidunmoye, F. Hernandez-Rodriguez, and E. Elmroth. Performance Anomaly Detection and Bottleneck Identification. ACM Computing Surveys, Vol. 48, No. 1, Article 4, p. 1-35, ACM. September 2015.
41. D. Sun, M. Fu, L. Zhu, G. Li, and Q. Lu. Non-intrusive anomaly detection with streaming performance metrics and logs for devops in public clouds: A case study in aws. In IEEE Transactions on Emerging Topics in Computing, vol. 4, no. 2, pp. 278–289. June 2016.
42. G. M. Abdelrahman and M. M. Nasr. Detection of Performance Anomalies in Cloud Services: A Correlation Analysis Approach. International Journal of

Mechanical Engineering and Information Technology, vol. 4, no. 9, pp. 1773–1781. 2016.

43. Daniel Sun, Min Fu, Liming Zhu, Guoqiang Li, Qinghua Lu. Non-Intrusive Anomaly Detection With Streaming Performance Metrics and Logs for DevOps in Public Clouds: A Case Study in AWS. In IEEE Transactions on Emerging Topics in Computing, vol. 4, no. 2, DOI: 10.1109/TETC.2016.2520883. IEEE. 2016.
44. Imam Yagoub, Muhammad Asif Khan, Li Jiyun. IT Equipment Monitoring and Analyzing System for Forecasting and Detecting Anomalies in Log Files Utilizing Machine Learning Techniques. In 2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD), DOI: 10.1109/ICABCD.2018.8465400. IEEE. 2018.
45. Qingfeng Du, TiandiXie, Yu He. Anomaly Detection and Diagnosis for Container-Based Microservices with Performance Monitoring. In International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'18), DOI:10.1007/978-3-030-05063-4_42. 2018.
46. C. Sauvanaud, M. Kaaniche, K. Kanoun, K. Lazri, and G. Da Silva Silvestre. Anomaly detection and diagnosis for cloud services: Practical experiments and lessons learned. In Journal of Systems and Software, vol. 139, pp. 84–106. 2018.
47. SashoNedelkoski TU Berlin, Jorge Cardoso, Odej Kao. Anomaly Detection and Classification using Distributed Tracing and Deep Learning. In 2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), DOI: 10.1109/CCGRID.2019.00038, IEEE. 2019.
48. O. Vallis, J. Hochenbaum, and A. Kejariwal. A novel technique for long-term anomaly detection in the cloud. In Proceedings of the 6th USENIX Conference on Hot Topics in Cloud Computing, ser. HotCloud'14. Berkeley, CA, USA: USENIX Association, pp. 15-15. 2014.
49. P. Malhotra, L. Vig, G. Shroff, and P. Agarwal. Long short term memory networks for anomaly detection in time series. In ESANN 2015 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, Bruges (Belgium), pp. 89-94. 2015.
50. Haowen Xu, Wenxiao Chen, Nengwen Zhao, Zeyan Li, Jiahao Bu, Zhihan Li, Ying Liu, Youjian Zhao, Dan Pei and Yang Feng, Jie Chen, Zhaogang Wang, Honglin Qiao. Unsupervised Anomaly Detection via Variational Auto-Encoder for Seasonal KPIs in Web Applications. In WWW 2018: The 2018 Web Conference, April 23–27, 2018, Lyon, France. ACM, New York, NY, USA, 12 pages. https://doi.org/https://doi.org/10.1145/3178876.3185996. 2018.
51. Run-Qing Chen, Guang-Hui Shi, Wan-Lei Zhao, Chang-Hui Liang. Sequential VAE-LSTM for Anomaly Detection on Time Series. ArXiv, abs/1910.03818. 2019.
52. Leandro von Werra, Lewis Tunstall, Simon Hofer. Unsupervised Anomaly Detection for Seasonal Time Series. In 2019 6th Swiss Conference on Data Science (SDS), DOI: 10.1109/SDS.2019.00036. IEEE. 2019.
53. Tian Huang, Yan Zhu, Qiannan Zhang, Yongxin Zhu, Dongyang Wang, MeikangQiu, and Lei Liu. An LOF-based adaptive anomaly detection scheme for

cloud computing. In Proceedings of the IEEE 37th Annual Computer Software and Applications Conference Workshops (COMPSACW'13). IEEE, pp. 206–211. 2013.

54. Tao Wang, Wenbo Zhang, Jun Wei, and Hua Zhong. Workload-aware online anomaly detection in enterprise applications with local outlier factor. In Proceedings of the IEEE 36th Annual Computer Software and Applications Conference (COMPSAC'12). IEEE, 25–34. 2012.
55. M. M. Breunig, H.-P. Kriegel, RRaymond T. Ng, and J. Sander. LOF: Identifying density-based local outliers. In Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 93–104, https://doi.org/10.1145/342009.335388. 2000.
56. Mohsin Munir, Shoaib Ahmed Siddiqui, Andreas Dengel, Sheraz Ahmed. DeepAnT: A Deep Learning Approach for Unsupervised Anomaly Detection in Time Series. In IEEE Access, Vol. 7, DOI: 10.1109/ACCESS.2018.2886457. 2018.
57. Lazarevic A, Ertoz L, Kumar V., Ozgur A, Srivastava J. A comparative study of anomaly detection schemes in network intrusion detection. In Proceedings of the 2003 SIAM International Conference on DataMining, San Francisco, CA, USA, Vol. 3, pp. 25–36. 2003.
58. James M. Keller, Michael R. Gray, James A. Givens. A fuzzy k-nearest neighbor algorithm. In IEEE Transactions on Systems, Man and Cybernetics, Vol. SMC-15, no. 4, 1985.
59. Kanishka Bhaduri, Kamalika Das, and Bryan L. Matthews. Detecting abnormal machine characteristics in cloud infrastructures. In Proceedings of the IEEE 11th International Conference on Data Mining Workshops (ICDMW'11). IEEE, pp. 137–144, DOI: 10.1109/ICDMW.2011.62. 2011.
60. Areeg Samir, Claus Pahl. DLA: Detecting and Localizing Anomalies in Containerized Microservice Architectures Using Markov Models. In 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud). DOI: 10.1109/FiCloud.2019.00036. IEEE. 2019.
61. Xiao Zhang, Fanjing Meng, Pengfei Chen, Jingmin Xu. TaskInsight: A Fine-Grained Performance Anomaly Detection and Problem Locating System. In 2016 IEEE 9th International Conference on Cloud Computing (CLOUD). DOI: 10.1109/CLOUD.2016.0136, IEEE. 2016.
62. Uzziah Eyee. "Microservices Observability with Distributed Tracing". https://medium.com/swlh/microservices-observability-with-distributed-tracing-32ae467bb72a, 2019. Accessed 2020-02-21.
63. Matthew McKenzie. "Complexity in Context: Microservices and Distributed Tracing [Video]". https://blog.newrelic.com/product-news/complexity-context-microservices-distributed-tracing-video/, 2019. Accessed 2020-02-21.
64. Open Tracing, "What is Distributed Tracing?". https://opentracing.io/docs/overview/what-is-tracing/. Accessed 2020-02-21.
65. Lightstep, "Lightstep Distributed Tracing". https://lightstep.com/distributed-tracing/. Accessed 2020-02-21.

66. Rtfpessoa. Understanding microservices with distributed tracing. https://blog.codacy.com/understanding-microservices-with-tracing/, 2019. Accessed 2020-02-21.
67. OpenCensus, “Tracing”. https://opencensus.io/tracing/. Accessed 2020-02-21.
68. RAN RIBENZAFT. Microservices: Using Distributed Tracing for Monitoring & Troubleshooting. https://cloudacademy.com/blog/microservices-using-distributed-tracing-monitoring-troubleshooting/, 2019. Accessed 2020-02-21.
69. SignalFx, “APM PG Instrumentation Overview”. https://docs.signalfx.com/en/latest/apm/apm-instrument/apm-instr-overview.html. Accessed 2020-02-21.
70. Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Technical Report dapper-2010-1, Google, Inc., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf
71. Juraci Paixão Kröhling. A guide to the open source distributed tracing landscape. https://developers.redhat.com/blog/2019/05/01/a-guide-to-the-open-source-distributed-tracing-landscape/, 2019. Accessed 2020-02-21.
72. RedHat, “What is Jaeger?”. https://www.redhat.com/en/topics/microservices/what-is-jaeger. Accessed 2020-02-21.
73. Gerred Dillon, “Intro to OpenTracing and OpenCensus for Distributed Tracing”. https://blog.appoptics.com/intro-to-opentracing-and-opencensus-for-distributed-tracing/, 2018. Accessed 2020-02-21.
74. Juraci Paixão Kröhling, “Distributed tracing in a microservices world”. https://opensource.com/article/18/9/distributed-tracing-microservices-world, 2018. Accessed 2020-02-21.
75. https://en.wikipedia.org/wiki/Neural_network. Accessed 2020-02-21.
76. Akash Singh. Anomaly detection for temporal data using long short-term memory (lstm), 2017.
77. PathMind, “A Beginner's Guide to Neural Networks and Deep Learning”. https://pathmind.com/wiki/neural-network. Accessed 2020-02-26.
78. TutorialsPoint, “Artificial Neural Network - Basic Concepts”, https://www.tutorialspoint.com/artificial_neural_network/artificial_neural_network_basic_concepts.htm. Accessed 2020-02-26.
79. Oleksii Trekhleb. https://github.com/trekhleb/machine-learning-octave/blob/master/neural-network/README.md. Accessed 2020-02-26.
80. Deval Shah, “Activation Functions”. https://towardsdatascience.com/activation-functions-in-neural-networks-58115cda9c96, April 2017. Accessed 2020-02-26.
81. MissingLink, “7 Types of Neural Network Activation Functions: How to Choose?”. https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/. Accessed 2020-02-26.

82. David Fumo, “A Gentle Introduction To Neural Networks Series — Part 1”. https://towardsdatascience.com/a-gentle-introduction-to-neural-networks-series-part-1-2b90b87795bc, August 2017. Accessed 2020-02-26.
83. Jordi TORRES.AI, “Learning process of a neural network”. https://towardsdatascience.com/how-do-artificial-neural-networks-learn-773e46399fc7, September 2018. Accessed 2020-02-26.
84. Mahendran Venkatachalam, “Recurrent Neural Networks”, https://towardsdatascience.com/recurrent-neural-networks-d4642c9bc7ce, February 2019. Accessed 2020-02-26.
85. Aditi Mittal, “Understanding RNN and LSTM”. https://towardsdatascience.com/understanding-rnn-and-lstm-f7cdf6dfc14e, October 2019. Accessed 2020-02-28.
86. Pranjal Srivastava, “Essentials of Deep Learning: Introduction to Long Short Term Memory”. https://www.analyticsvidhya.com/blog/2017/12/fundamentals-of-deep-learning-introduction-to-lstm/, December 2017. Accessed 2020-02-28.
87. SuperDataScience Team, “Recurrent Neural Networks (RNN) - The Vanishing Gradient Problem”. https://www.superdatascience.com/blogs/recurrent-neural-networks-rnn-the-vanishing-gradient-problem, August 2018. Accessed 2020-03-04.
88. Christopher Olah. “Understanding LSTM Networks”. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, August 2015. Accessed 2020-03-04.
89. Devby RayRay, “GraphQL (microservices) architecture by Apollo”. https://itnext.io/graphql-microservices-architecture-by-apollo-8b6eb557c5e2, January 15. Accessed 2020-03-07.
90. HOWTOGRAPHQL, “GraphQL is the better REST”. https://www.howtographql.com/basics/1-graphql-is-the-better-rest/. Accessed 2020-03-07.
91. Zhao Yue, Nasrullah Zain, Hryniewicki Maciej, Li Zheng. LSCP: Locally Selective Combination in Parallel Outlier Ensembles. 2019.
92. Vadim Smolyakov, “Ensemble Learning to Improve Machine Learning Results”. https://blog.statsbot.co/ensemble-learning-d1dcd548e936, August 2017. Accessed 2020-03-10
93. Tin Kam Ho, J.J. Hull, S.N. Srihari. Decision Combination in Multiple Classifier Systems. In IEEE Transactions on Pattern Analysis and Machine Intelligence. Vol 16, no 1, DOI: 10.1109/34.273716. Jan 1994.
94. Singh, Archana & Yadav, Avantika & Rana, Ajay. K-means with Three different Distance Metrics. International Journal of Computer Applications. Vol. 67 no. 10, pp 13-17, DOI:10.5120/11430-6785. 2013.
95. GeeksforGeeks, “Elbow Method for optimal value of k in KMeans”. https://www.geeksforgeeks.org/elbow-method-for-optimal-value-of-k-in-kmeans/. Accessed 2020-03-16.

96. Venkatesh Umamaheswaran, “Comprehending K-means and KNN Algorithms”. https://becominghuman.ai/comprehending-k-means-and-knn-algorithms-c791be90883d, November 2018. Accessed 2020-03-16.
97. https://pyod.readthedocs.io/en/latest/. Accessed 2020-03-16.
98. Manojit Nandi, “LOCAL OUTLIER FACTOR – PART 1”. https://blog.stealthbits.com/local-outlier-factor-part-1. Accessed 2020-03-16.
99. Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jorg Sander. LOF: identifying density-based local outliers. In ACM sigmod record, Vol 29, pp 93-104. 2000.
100. Fei Tony Liu, Kai Ming Ting, Zhi-Hua Zhou. Isolation Forest. In 2008 Eighth IEEE International Conference on Data Mining. DOI: 10.1109/ICDM.2008.17. 2008.
101. Hyunsu Kim, "Isolation Forest Step by Step". https://medium.com/@hyunsukim_9320/isolation-forest-step-by-step-341b82923168, January 2019. Accessed 2020-03-16.
102. Schölkopf Bernhard, Williamson Robert, Smola Alex, Shawe-Taylor John, Platt John. Support Vector Method for Novelty Detection. In NIPS'99: Proceedings of the 12th International Conference on Neural Information Processing Systems. 1999.
103. V.K. Chouksey. Automatic selection of outlier detection techniques. 2018.
104. https://en.wikipedia.org/wiki/Cloud_computing
105. "Cloud Computing Technologies: A Global Outlook". https://www.businesswire.com/news/home/20190708005357/en/Cloud-Computing-Technologies-Global-Outlook-2019--
106. https://stats.stackexchange.com/questions/282987/hidden-markov-model-vs-recurrent-neural-network. Accessed 2020-05-19.
107. https://www.codecademy.com/articles/normalization. Accessed 2020-05-19.
108. https://numenta.com/machine-intelligence-technology/numenta-anomaly-benchmark/. Accessed 2020-03-01.
109. https://github.com/numenta/NAB/blob/master/results/numenta/numenta_standard_scores.csv. Accessed 2020-05-23.
110. https://www.synextra.co.uk/evolution-cloud-computing/. Accessed 2020-05-27.

Appendix A

A.1 Data Extraction Algorithms

A.1.1 Algorithm to extract Trace IDs from Elasticsearch system

---

```
#!/usr/bin/env python
import urllib3
import certifi
import pandas as pd
import pickle
import json
import time
from datetime import datetime, date, timedelta
from elasticsearch import Elasticsearch

def datetime_range(start, end, delta):
   current = start
   while current < end:
      yield current
      current += delta

dts = [dt.strftime('%Y-%m-%dT%H:%M:%S') for dt in
     datetime_range(datetime(2019, 8, 30, 23, 55), datetime(2019, 9, 2, 0),
     timedelta(minutes=5))]
credentials = ""
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
es = Elasticsearch(['https://priyanka.naikade@****.com:******@bss-*******-
p00.compose.direct:15117/'], verify_certs=False )

#get trace IDs of all the graphql requests. max len will be 10,000 due to size
parameter
traceinfolist = []
trIDs = []

for i in dts:
   from_time = i
   to_time = datetime.strptime(i, '%Y-%m-%dT%H:%M:%S') +
timedelta(minutes=5)
   res3 = es.search(index="cloud-datalayer*", body = {
      'size' : 10000,
      'query': {
      'bool': {
```

```
    'filter': [
     {
      'range': {
       'date': {
        'from': from_time,
        'to': to_time
       }
      }
     },
     {
      'term': {
       'name.keyword': {
        'value': '/datalayer/graphql'
       }
      }
     }
    ]
   }
  }
   })

for doc in res3['hits']['hits']:
   trIDs.append(doc['_source']['traceId'])

picklefilename = "trIDs"+str(to_time)+".pickle"
pickle_out = open(picklefilename,"wb")
pickle.dump(trIDs, pickle_out)
pickle_out.close()
print("*****************************")
time.sleep( 5 )
traceinfolist = []
trIDs = []
print("\n\n-----------------------------------end----------------------------------------")
```

A.1.2 Algorithm to get trace information for the collected Trace IDs

```
#!/usr/bin/env python
import urllib3
import certifi
import pandas as pd
import pickle
import json
import time
#import simplejson
import time
from datetime import datetime, date, timedelta
import os
from elasticsearch import Elasticsearch
import smtplib
from smtplib import SMTPException


start = time.time()
credentials = ""
urllib3.disable_warnings(urllib3.exceptions.InsecureRequestWarning)
es = Elasticsearch(['https://priyanka.naikade@****.com:******@bss-
*******p00.compose.direct:15117/'], verify_certs=False )

#Slicing list of trace IDs for max size limit issue while fetching data from
Elasticsearch
def slicelist(listdata, num):
   avg = len(listdata) / float(num)
   print(avg)
   outlistset = []
   last = 0.0
   while last < len(listdata):
      outlistset.append(listdata[int(last):int(last + avg)])
      last += avg
   return outlistset

# Get the directory path from the user where all the input files are present
dirpath = "/home/priyanka/thesis/ESdata/tidfiles/"
files = os.listdir(dirpath)
# Looping through every file present in the folder and storing its information in a
list
for each_file in files:
   new_filename = "Info" + each_file
```

```
   #get trace IDs of all the graphql requests. max len will 10,000 due to size
parameter
   traceinfolist = []
   trIDs = []


   filepath = "/home/priyanka/thesis/ESdata/tidfiles/" + each_file
   pickle_trlist = open(filepath,"rb")
   trlist_orig = pickle.load(pickle_trlist)
   transactions_dict = {}
   trans_list = []
   troutlist = slicelist(trlist_orig,20)
   tinf =[]

   for trlist in troutlist:
      data = es.search(index="cloud-datalayer*", body = {
         'size' : 10000,
         'query': {
         'bool': {
          'filter': [
              {
               'terms':{
                    'traceId.keyword': trlist

                 }
              }
             ]
         }
       }})

      ''',
          'sort': [
           { 'traceId.keyword':   { "order": "asc" }}
          ]'''

      for doc in data['hits']['hits']:
         tinf.append((doc['_source']))
         key = doc['_source']['traceId']
         if key in transactions_dict.keys():
            #parent already present so appendd child values to the existing key in the
dictionary")
            transactions_dict[key].append((doc['_source']))
         else:
            #new parent so add in the dictionary")
            transactions_dict[key] = [(doc['_source'])]
```

```
    print("\n writing to files\n")
    pickle_out = open(new_filename,"wb")
    pickle.dump(transactions_dict, pickle_out)
    pickle_out.close()
    print(new_filename,"done writing")

end = time.time()
ttaken = end -  start
print("\n time taken:",ttaken)

#Send an email notification
to = 'priyanka*****@gmail.com'
gmail_user = 'priyanka*****@gmail.com'
gmail_pwd = '******'
smtpserver = smtplib.SMTP("smtp.gmail.com",587)
smtpserver.ehlo()
smtpserver.starttls()
smtpserver.ehlo
smtpserver.login(gmail_user, gmail_pwd)
header = 'To:' + to + '\n' + 'From: ' + gmail_user + '\n' + 'Subject: ES files extracted
\n'
#print(header)
msg = header + '\n Trace Info Objects from ES extracted \n\n time taken : ' +
str(ttaken)
smtpserver.sendmail(gmail_user, to, msg)
print('done!')
smtpserver.close()
```

A.1.3 Algorithm to get request summary information

```
#Get requests Summary from ES files along with response codes
import os
import pickle
import json
import requests
import time
from datetime import datetime, date, timedelta
from dateutil.parser import parse

dirpath = '/mnt/c/Users/Priyanka Naikade/trinfoES/'
tags = ('traceId','date','duration')
newtags = ('traceId','name', 'id', 'parentId','date', 'timestamp',
'duration','http.host','http.status_code')
files = os.listdir(dirpath)
graphql_time = {}
v ={}
cd={}
graphql_timelist = []
codes=[]

# Looping through every file present in the folder and storing its information in a
list
for each_file in files:
   c=0
   total = 0
   codes = []
   filepath =  dirpath + each_file
   pickle_trinf = open(filepath,"rb")
   trinfo = pickle.load(pickle_trinf)
   for t in trinfo:
      #t is key trace ID, trinfo list
      #for every transaction we need only certain fields like ID, parentID, duration for
graph creation
      #so removed some key fields from the inner dictionary
      for span in trinfo[t]:
         #print(span['name'],":",span['duration'])
         if(span['name'] == '/datalayer/graphql'):
            #graphql_time[span['name']] = span['duration']
            t = span['duration']/1000
            total = total + t

         #get all response codes in list
```

```
        for tagname in span['tags']:
            #tagname - tagnames
            if(tagname == 'http.status_code'):
                #print(span['tags']['http.status_code'])
                codes.append(span['tags']['http.status_code'])

    #get count for every distinct response code
    cde=0
    for i in set(codes):
        for rc in codes:
            if i == rc:
                cde=cde+1

        cd[i] = cde
        cde = 0


    #print(each_file,":",total)
    date = (each_file[9:28])
    graphql_time["k"]= date
    v["count"]= len(trinfo)
    v["duration"]=total
    graphql_time["v"]=v
    graphql_time["rescodes"]=cd
    graphql_timelist.append((graphql_time))
    graphql_time = {}
    v ={}
    cd ={}

pickle_out = open("/home/priyanka/ESrequestSummarywithcodes2.pickle","wb")
pickle.dump(graphql_timelist, pickle_out)
pickle_out.close()
print("end")
```

A.1.4 Algorithm to get Individual summary information

```
import os
import requests
import json
import pickle

#get Individual Summary --> requests, total duration of cache, get, post, every 5mins
interval
dirpath = "/mnt/c/Users/Priyanka Naikade/trinfoES/"
tags = ('traceId','name', 'id', 'parentId','date','timestamp', 'duration','tags')
newtags = ('traceId','name', 'id', 'parentId','date', 'timestamp',
'duration','http.host','http.status_code')

files = os.listdir(dirpath)
IndividualSummary = {}
v={}
IndividualSummarylist =[ ]
# Looping through every file present in the folder and storing its information in a
list
for each_file in files:
   t, ct, pt, gt, ot, total = 0,0,0,0,0,0
   tgraph, GRAPH, CACHE, tcache, POST, tpost, tget, GET ,othercategory, tother=[ ],[
],[ ],[ ],[ ],[ ],[ ],[ ],[ ],[ ]
   cacheduration, cacherequests, postduration, postrequests, getduration,
getrequests, graphqlduration, otherduration,otherrequests = 0,0,0,0,0,0,0,0,0
   filepath =  dirpath + each_file
   pickle_trinf = open(filepath,"rb")
   trinfo = pickle.load(pickle_trinf)
   for t in trinfo:
      #t is key trace ID, trinfo list
      #for every transaction we need only certain fields like ID, parentID, duration for
graph creation
      #so removed some key fields from the inner dictionary
      for span in trinfo[t]:
         #print(span['name'],":",span['duration'])
         if(span['name'] == '/datalayer/graphql'):
            #graphql_time[span['name']] = span['duration']
            t = span['duration']/1000
            tgraph.append(t)
         elif(span['name'].startswith('CACHE')):
            ct = span['duration']/1000
```

```
            tcache.append(ct)
        elif(span['name'].startswith('POST')):
            pt = span['duration']/1000
            tpost.append(pt)
        elif(span['name'].startswith('GET')):
            gt = span['duration']/1000
            tget.append(gt)
        else:
            ot = span['duration']/1000
            tother.append(ot)

    GRAPH.extend(tgraph)
    CACHE.extend(tcache)
    POST.extend(tpost)
    GET.extend(tget)
    othercategory.extend(tother)
    graphduration= sum(GRAPH)
    graphrequests = len(GRAPH)
    cacheduration = sum(CACHE)
    cacherequests = len(CACHE)
    postduration = sum(POST)
    postrequests = len(POST)
    getduration = sum(GET)
    getrequests = len(GET)
    otherduration = sum(othercategory)
    otherrequests = len(othercategory)

    date = (each_file[9:28])
    IndividualSummary["k"]= date
    v["graphRequests"]= graphrequests
    v["graphDuration"]= graphduration
    v["cacheRequests"]= cacherequests
    v["cacheDuration"]= cacheduration
    v["postRequests"]= postrequests
    v["postDuration"]= postduration
    v["getRequests"]= getrequests
    v["getDuration"]= getduration
    v["otherRequests"]= otherrequests
    v["otherDuration"]= otherduration
    IndividualSummary["v"]=v
    IndividualSummarylist.append((IndividualSummary))
    IndividualSummary = {}
    v ={}

pickle_out = open("/home/priyanka/IndividualSummarylist2.pickle","wb")
pickle.dump(IndividualSummarylist, pickle_out)
```

```
pickle_out.close()
```

A.1.5 Algorithm to get all dates of Traces

```
import requests
import os
import pickle
import json

dirpath = "/mnt/c/Users/Priyanka Naikade/trinfoES/"
files = os.listdir(dirpath)
# Looping through every file present in the folder and storing its information in a
list
Date_traceIDs = {}
for each_file in files:
   filepath =  dirpath + each_file
   pickle_trinf = open(filepath,"rb")
   trinfo = pickle.load(pickle_trinf)
   fname = each_file[9:-7]
   traces = []
   for t in trinfo:
      traces.append(t)
   Date_traceIDs[fname]=traces

pickle_out = open("/home/priyanka/AllDate_traceIDs2.pickle","wb")
pickle.dump(Date_traceIDs, pickle_out)
pickle_out.close()

print("End")
```

A.2 Localization Module – Mapper Function

```
def mapper(df_test, cachedf_test, postdf_test, getdf_test):
        df_test.loc[df_test['lscpoutlier_3']==1, 'Date'].tolist()
        graphqlanomalies = []
        graphqlanomalies = df_test.loc[df_test['lscpoutlier_3']==1, 'Date'].tolist()
        cacheanomalies = []
        cacheanomalies = cachedf_test.loc[cachedf_test['lscpoutlier_more']==1,
'Date'].tolist()
        postanomalies = []
        postanomalies = postdf_test.loc[postdf_test['lscpoutlier_more']==1, 'Date'].tolist()
        getanomalies = []
        getanomalies = getdf_test.loc[getdf_test['lscpoutlier_more']==1, 'Date'].tolist()
        cdict = {}
        cano = []
        for i in graphqlanomalies:
                if i in cacheanomalies:
                        print(i)
                        cano.append(i)

        cdict['cache'] = cano
        pdict = {}
        pano = []
        for i in graphqlanomalies:
                if i in postanomalies:
                        #print(i)
                        pano.append(i)

        pdict['post']= pano
        gedict = {}
        geano = []
        for i in graphqlanomalies:
                if i in getanomalies:
                        #print(i)
                        geano.append(i)

        gedict['get']= geano
        anodict = {}
        for i in graphqlanomalies:
                if (i in cacheanomalies) & (i in postanomalies) & (i in getanomalies):
                        print(i)
                        anodict[i] = ['cache','post','get']
                elif (i in cacheanomalies) & (i in postanomalies) & (i not in getanomalies):
                        #print(i)
                        anodict[i] = ['cache','post']
```

```
                elif (i in cacheanomalies) & (i not in postanomalies) & (i not in
getanomalies):
                        anodict[i] = ['cache']
                elif (i not in cacheanomalies) & (i in postanomalies) & (i in getanomalies):
                        anodict[i] = ['post','get']
                elif (i not in cacheanomalies) & (i not in postanomalies) & (i in
getanomalies):
                        anodict[i] = ['get']
                elif (i not in cacheanomalies) & (i in postanomalies) & (i not in
getanomalies):
                        anodict[i] = ['post']
                elif (i in cacheanomalies) & (i not in postanomalies) & (i in getanomalies):
                        anodict[i] = ['cache','get']
                elif (i not in cacheanomalies) & (i not in postanomalies) & (i not in
getanomalies):
                        anodict[i] = ['none']

        #generate color request flow graph for each anomalous interval and visualize
using graphs
        for key,val in anodict.items():
                print("\nAnomalous Interval:",key,"\nCausal Component red-highlighted
in graph")
                G = nx.DiGraph()
                map(G.add_node, range(6))

                #Add all nodes along with their color information.
                #Datalayer-black, graphql-green, Cache- Blue, POST-cyan, GET- red
                k = ['Dashboard','Datalayer','GraphQL','POST','GET','CACHE']
                G.add_node('Dashboard',pos=(1,3),color='yellowgreen')
                G.add_node('Datalayer',pos=(2,3),color='yellowgreen')
                G.add_node('GraphQL',pos=(3,3),color='yellowgreen')

                if val == ['none']:
                        G.add_node('POST',pos=(4,3),color='yellowgreen')
                        G.add_node('GET',pos=(4,4),color='yellowgreen')
                        G.add_node('CACHE',pos=(4,2),color='yellowgreen')
                elif val == ['cache','post','get']:
                        G.add_node('POST',pos=(4,3),color='red')
                        G.add_node('GET',pos=(4,4),color='red')
                        G.add_node('CACHE',pos=(4,2),color='red')
                elif val == ['cache','post']:
                        G.add_node('POST',pos=(4,3),color='red')
                        G.add_node('GET',pos=(4,4),color='yellowgreen')
                        G.add_node('CACHE',pos=(4,2),color='red')
                elif val == ['cache']:
                        G.add_node('POST',pos=(4,3),color='yellowgreen')
```

```
                G.add_node('GET',pos=(4,4),color='yellowgreen')
                G.add_node('CACHE',pos=(4,2),color='red')
        elif val == ['post','get']:
                G.add_node('POST',pos=(4,3),color='red')
                G.add_node('GET',pos=(4,4),color='red')
                G.add_node('CACHE',pos=(4,2),color='yellowgreen')
        elif val == ['post']:
                G.add_node('POST',pos=(4,3),color='red')
                G.add_node('GET',pos=(4,4),color='yellowgreen')
                G.add_node('CACHE',pos=(4,2),color='yellowgreen')
        elif val == ['get']:
                G.add_node('POST',pos=(4,3),color='yellowgreen')
                G.add_node('GET',pos=(4,4),color='red')
                G.add_node('CACHE',pos=(4,2),color='yellowgreen')
        elif val == ['cache','get']:
                G.add_node('POST',pos=(4,3),color='ywllowgreen')
                G.add_node('GET',pos=(4,4),color='red')
                G.add_node('CACHE',pos=(4,2),color='red')


        G.add_edge('Dashboard','Datalayer')
        G.add_edge('Datalayer','GraphQL')
        G.add_edge('GraphQL','POST')
        G.add_edge('GraphQL','GET')
        G.add_edge('GraphQL','CACHE')
        pos=nx.get_node_attributes(G,'pos')
        red_nodes=[n for n,d in G.nodes(data=True) if d['color']=='red'] #graphql
        yellowgreen_nodes=[n for n,d in G.nodes(data=True) if
d['color']=='yellowgreen'] #cache

        #draw respective color category nodes

    nx.draw(G,pos,alpha=0.7,nodelist=yellowgreen_nodes,node_color='yellowgreen',
node_size=4500,node_shape='s')

    nx.draw(G,pos,alpha=0.7,nodelist=red_nodes,node_color='red',node_size=4500,n
ode_shape='s')
        nx.draw_networkx_labels(G,pos,font_color='b')
        nx.draw_networkx_edges(G,pos, width=1,arrowsize=10)
        plt.figure(figsize=(30, 20))
        plt.show(block=False)


mapper(df_test, cachedf_test, postdf_test, getdf_test)
```

A.3 Algorithm for Information Module

To display Summary of a transaction, request flow diagram for a given trace ID entered by the user

```
#code to fetch information/summary for the input trace ID - option 4
def GenerateListsforSingleTrace(SingleTraceInfoList):
   tags = ('date','timestamp','traceId','http.status_code','http.host')
   #includedtags = ('id','parentId','duration')
   #for every transaction we need only certain fields like ID, parentID, duration for
graph creation
   #so removed some key fields from the inner dictionary
   original_list=SingleTraceInfoList
   new_list = [{k: v for k, v in d.items() if k not in tags} for d in original_list]
   #overwrite dictionary value i.e value for trace ID
   SingleTraceInfoList=new_list

   #create a list of dictinaries which has info of individual spans [ ID1 : parent,
duration, ID2: parent, duration..]
   spans_list = []
   eachspan ={}
   #Get total number of nodes involved in a transaction and their IDs
   listofnodes=[]
   listofnodes2 =[]
   #Get Node & their duration given in json objects which included chil's duration as
well
   Id_Duration_given = {}

   #Modified list ---looping through the list of spans within the transaction
   for s in SingleTraceInfoList:
      listofnodes.append(s['id'])
      listofnodes.append(s['parentId'])
      eachspan[s['id']]=s
      Id_Duration_given[s['id']]=s['duration']

   spans_list.append(eachspan)

   NumofNodes = len(list(set(listofnodes)))
   #print("ID-duration:",Id_Duration_given)

   #Iterate through spans_list
```

```
   #Create a dictionary for graph -- {parent1:[child1, child2], parent2: [child1]}
   parentchild = {}
   for spans in spans_list:
      #looping through the list of spans within the transaction
      for each in spans:
         #print(each) #each is key-ID, spans[each]=values -duration,parentID
         #print("Parent",spans[each]['parentId'],":","ID",each)
         pkey = spans[each]['parentId']
         child = each
         #if key/parent ID doesn't exist
         if pkey in parentchild.keys():
            #parent already present so appendd child values to the existing key in the
dictionary")
            parentchild[pkey].append(child)
         else:
            #new parent so add in the dictionary")
            parentchild[pkey] = [child]
   return NumofNodes,list(set(listofnodes)), spans_list, parentchild,
Id_Duration_given


#Get the individual node's duration because the json adds up the duration of child
node & CPU to the parent node
#This has considered only child nodes duration and has ignored the CPU/network
duration or time gaps between the spans
def GetIndividualNodesDuration(Id_Duration_given,parentchild):
   #loop through ID-duration because it starts from graphql node and not the nodes
before graphql/which calls graphQl
   ID_individual_duration = {}
   total_child_duration = 0
   for idd in Id_Duration_given:
      #print(idd,Id_Duration_given[idd])
      #check if the node has any child --i.e., check if it's a parent/key in parentchild
dictionary
      if idd in parentchild.keys():
         #check how many childs and calculate the sum of duration of all its childrens
         #print(parentchild[idd])
         for c in parentchild[idd]:
            total_child_duration += Id_Duration_given[c]

         pduration = Id_Duration_given[idd] - total_child_duration
         # add to dictionary
         ID_individual_duration[idd]=pduration/1000

      else:
```

```
        #Set the given duration as it's individual duration since it doesn't have any
child
        ID_individual_duration[idd]=Id_Duration_given[idd]/1000

    return ID_individual_duration


#call the respective functions for every transaction information

def GetSingleTransactionInfo(single_transaction):
    NumofNodes, listofnodes, spans_list, parentchild, Id_Duration_given =
GenerateListsforSingleTrace(single_transaction)
    ID_individual_duration =
GetIndividualNodesDuration(Id_Duration_given,parentchild)

    return NumofNodes, listofnodes, spans_list, parentchild,
Id_Duration_given,ID_individual_duration


def CreateCategoryBuckets(ID_individual_duration, spans_list):
    #creating buckets to append timestamps of graphql, chache, post
    # use spanlist, ID_individual_duration list to get the names&time for grouping into
respective buckets
    # ID_individual_duration is adictionary
    tCACHE = []
    tPOST = []
    tGRAPHQL = []
    tGET = []
    tothercategory = []
    nodedict = {}
    nodecolors = {}
    for kid, v in ID_individual_duration.items():
        #print(kid,v)
        for s in spans_list:
            for i in s:
                if (kid == i):
                    #print("\n name of kid:",i,":",s[i]['name'])

                    if(s[i]['name'] == '/datalayer/graphql'):
                        namepart = "GraphQL-"
                        nodecolors[kid] = "red"
                    elif(s[i]['name'].startswith('CACHE')):
                        namepart = "CACHE-"
                        nodecolors[kid] = "purple"
                    elif(s[i]['name'].startswith('POST')):
                        namepart = "POST-"
```

```
                nodecolors[kid] = "blue"
            elif(s[i]['name'].startswith('GET')):
                namepart = "GET-"
                nodecolors[kid] = "green"
            else:
                namepart = ""
                nodecolors[kid] = "cyan"


            nodedict[namepart+kid] = v

    #print("\n nodedict:", nodedict)
    for k,v in nodedict.items():
        if(k.startswith('CACHE')):
            tCACHE.append(v)
        elif(k.startswith('POST')):
            tPOST.append(v)
        elif(k.startswith('GET')):
            tGET.append(v)
        elif(k.startswith('GraphQL')):
            tGRAPHQL.append(v)
        else:
            tothercategory.append(v)

    return nodedict, tCACHE, tPOST, tGET, tGRAPHQL, tothercategory,nodecolors


def drawrequestflowgraph(NumofNodes,parentchild,nodecolors):
    G = nx.DiGraph()
    map(G.add_node, range(NumofNodes))

    #Add all nodes along with their color information.
    #Datalayer-cyan, graphql-red, POST- Blue, CACHE-purple, GET- green
    for k,v in nodecolors.items():
        G.add_node(k,color = v)
    #Datalyer node is absent in nodecolors dictionary. hence mention black for the
root node explicitly
    for p in parentchild:
        if(p not in nodecolors.keys()):
            G.add_node(p,color='cyan')

    #iterate through list to find edges between nodes (from parent to child)
    for p in parentchild:
        print(p)
        for c in parentchild[p]:
            G.add_edge(p,c)
```

```
   pos = nx.spring_layout(G)
   pos2 = nx.kamada_kawai_layout(G)
   #G.nodes is a dictionary with keys: node name and color
   red_nodes=[n for n,d in G.nodes(data=True) if d['color']=='red'] #graphql
   purple_nodes=[n for n,d in G.nodes(data=True) if d['color']=='purple'] #cache
   blue_nodes=[n for n,d in G.nodes(data=True) if d['color']=='blue'] #POST
   green_nodes=[n for n,d in G.nodes(data=True) if d['color']=='green'] #GET
   cyan_nodes=[n for n,d in G.nodes(data=True) if d['color']=='cyan'] #Datalayer
root


   #draw respective color category nodes
   nx.draw_networkx_nodes(G,pos,nodelist=red_nodes,node_color='red',alpha=0.7)

nx.draw_networkx_nodes(G,pos,nodelist=purple_nodes,node_color='purple',alpha=
0.7)

nx.draw_networkx_nodes(G,pos,nodelist=blue_nodes,node_color='blue',alpha=0.7)

nx.draw_networkx_nodes(G,pos,nodelist=green_nodes,node_color='green',alpha=0.7
)

nx.draw_networkx_nodes(G,pos,nodelist=cyan_nodes,node_color='cyan',alpha=0.7)

   nx.draw_networkx_edges(G,pos)
   #nx.draw_networkx_labels(G,pos,font_color='w')
   plt.figure(figsize=(25, 15))
   plt.show(block=False)

    #draw respective color category nodes
   nx.draw_networkx_nodes(G,pos2,nodelist=red_nodes,node_color='red',alpha=0.7)

nx.draw_networkx_nodes(G,pos2,nodelist=purple_nodes,node_color='purple',alpha
=0.7)

nx.draw_networkx_nodes(G,pos2,nodelist=blue_nodes,node_color='blue',alpha=0.7)

nx.draw_networkx_nodes(G,pos2,nodelist=green_nodes,node_color='green',alpha=0.
7)

nx.draw_networkx_nodes(G,pos2,nodelist=cyan_nodes,node_color='cyan',alpha=0.7
)

   nx.draw_networkx_edges(G,pos2)
   #nx.draw_networkx_labels(G,pos,font_color='w')
```

```
    plt.figure(figsize=(25, 15))
    plt.show(block=False)


def lookupfortraceID(inputTraceID):
    traces = open("AllDate_traceIDs.pickle","rb")
    #traces = open("/home/priyanka/Date_traceIDs.pickle","rb")
    datetrace = pickle.load(traces)
    #s="9e82268b99b44ebc92d117ae73481fa8"
    checkdate = ""
    dictdata = {}
    listofdictdata = []
    for k,v in datetrace.items():
        #print(k,":",v)
        for i in range(len(v)):
            if(v[i]==inputTraceID):
                print(inputTraceID," present in", k)
                checkdate = k
    return checkdate

def lookupfortraceIDInfo(inputTraceID,checkdate):
    #dirpath = "/mnt/c/Users/Priyanka Naikade/trinfoES/"
    tags = ('traceId','name', 'id', 'parentId','date','timestamp', 'duration','tags')
    newtags = ('traceId','name', 'id', 'parentId','date', 'timestamp',
'duration','http.host','http.status_code')


    #files = os.listdir(dirpath)
    trinfocnt = {}
    tinfc = []
    trinfocount = []
    #trfilename = "InfotrIDs" + checkdate + ".pickle"

    pickle_trinf = open("InfotrIDs2019-06-27 13□10□00.pickle","rb")
    trinfo = pickle.load(pickle_trinf)
    for t in trinfo:
        if(t == inputTraceID):
            #t is key trace ID, trinfo list
            #for every transaction we need only certain fields like ID, parentID, duration
for graph creation
            #so removed some key fields from the inner dictionary
            original_list=trinfo[t]
            new_list = [{k: v for k, v in d.items() if k in tags} for d in original_list]
            for d in new_list:
                for tg in tags:
                    if tg not in d.keys():
```

```
                d[tg] = ""

        for d in new_list:
            if 'tags' in d.keys():
                if ('http.host') in d['tags'].keys():
                    d['http.host']= d['tags']['http.host']
                else:
                    d['http.host']= ""
                if ('http.status_code') in d['tags'].keys():
                    d['http.status_code']= d['tags']['http.status_code']
                else:
                    d['http.status_code']= ""

        new_list = [{k: v for k, v in d.items() if k in newtags} for d in new_list]

        #overwrite dictionary value i.e value for trace ID
        trinfo[t]=new_list
        singletraceinfo = trinfo[t]
        NumofNodes, listofnodes, spans_list, parentchild,
Id_Duration_given,ID_individual_duration =
GetSingleTransactionInfo(singletraceinfo)
        nodedict, tcache, tpost, tget, tgraphql, tothercategory,nodecolors =
CreateCategoryBuckets(ID_individual_duration, spans_list)

    return original_list, NumofNodes, parentchild, nodedict,nodecolors


topt = input("Choose the below options: \n1.Get Trace Information \n2.Go Back\n")
    if topt == "1":
        traceid = input("Enter trace ID:\n")
        checkdate = lookupfortraceID(traceid)
        if(checkdate == ""):
            print("Trace ID not found. Please enter a valid trace ID")
        else:
            original_list, NumofNodes, parentchild, nodedict,nodecolors =
lookupfortraceIDInfo(traceid,checkdate)
            while True:
                traceopt = input("Choose the below options: \n1.Trace Graph \n2.Trace
JSON \n3.Summary of Transaction \n4. Go Back \n")
                if traceopt == "1":
                    print("Request Flow Graph for the entered trace ID:",traceid,"\n")
                    print("\nParent-Child:", parentchild)
                    drawrequestflowgraph(NumofNodes,parentchild,nodecolors)
                elif traceopt == "2":
                    print("JSON for Trace ID:",traceid,"\n",original_list)
                elif traceopt == "3":
```

```
            print("\n**************************Summary of every single
transaction************************************")
            print("\nTransaction ID:",traceid,"Number of Nodes in
transaction:",NumofNodes)
            print("\nNodes and Time Taken by Nodes (in ms):\n",nodedict)
            print("\nParent-Child:", parentchild)
         elif traceopt == "4":
            break;
         else:
            print("Please enter a valid option!")
   elif topt == "2":
      break;
```

Curriculum Vitae

Name: Priyanka Naikade

Post-Secondary Education and Degrees:

SASTRA University
Thanjavur, Tamil Nadu, India
2010 - 2014
B.Tech. - Information and Communication Technology

The University of Western Ontario
London, Ontario, Canada
2018 - 2020
Master of Science - Computer Science

Honours and Awards:

Western Graduate Research Scholarship
2018 – 2019

Related Work Experience:

Graduate Research Assistant & Teaching Assistant
(Computer Science Department)
The University of Western Ontario
2018 - 2020

Software Developer Intern
IBM Software Lab
2019

Systems Engineer
Tata Consultancy Services
2014 - 2018